Analysis, Specification and Generation of Mobile Computer Data Synchronisation

Qingsong Ye, Anthony M. Sloane, and Dominic R. Verity
Department of Computing, Macquarie University, Sydney, NSW 2109, Australia
{qingsong, asloane, domv}@comp.mq.edu.au

Abstract

In current technologies, mobile computer data synchronisation protocols are typically programmed at a low-level. The disadvantages of this are that they are error-prone and time consuming. This paper analyses current Palm OS data synchronisation problems and applies embedded domain-specific language (EDSL) techniques in the problem domain to improve the current situation. The key advantage is that domain developers can describe problems using their natural terms and concepts at high conceptual level. Our approach allows equivalent code to be generated automatically from higher-level specifications, enabling domain developers to express their ideas quickly and concisely, to work more productively and to avoid certain kinds of coding error.

1. Introduction

Mobile computer data synchronisation ensures that the data on mobile and non-mobile computers are kept mutually consistent. Simple protocols implement 'overwrite' copying of data in either direction. More usefully, the two ends can be synchronised by a more complex protocol so that modifications at either end are reflected at the other [3].

In current technologies, data synchronisation protocols are typically programmed at a low level. In most cases, it is up to the programmer to explicitly develop algorithms for implementing an appropriate synchronisation policy [2]. With this approach there is no guarantee that a given piece of code actually implements desired policies. Furthermore, if the structure of the data is amended or the policy options are changed, much of the low-level coding will need to be redone. This method is error-prone and time consuming.

In this paper, we demonstrate our improvement on this situation by investigating formal specification methods for data synchronisation problems, particularly focusing on Palm OS handheld computers because of their prominent position in the mobile computing market. We apply embedded domain-specific language (EDSL) [5] techniques to develop a high-level interface using a higher-order, typed language, in this case Haskell [9]. Consequently, the domain developers have more flexibility and power to specify complex Palm OS conduits [3]. With our approach, the data synchronisation can be worked on a field-level conceptually, but only on a record-level currently. Due to space limitations, some of the detail in this paper assumes knowledge of Haskell but the essence of the method can be comprehended without a Haskell background.

By taking advantage of Haskell Prettyprinting techniques [8], the use of automatic code generation techniques is also investigated to increase productivity by capturing repetitive code sequences. This calls for some form of code reuse and expertise reuse [14]. The benefit of this reduces the software development time, which reduces the time-to-market. This gives a competitive advantage in the market for business related to this domain.

The remainder of this paper is organized as follows: Section 2 analyses the problem domain. The disadvantages of the current approach are discussed in Section 3. Section 4 introduces EDSL techniques as a high-level solution for this domain. In Section 5 the development process of the EDSL for Palm OS conduits is described. The EDSL designed for this domain is called Conduit EDSL. Section 6 puts all things together to illustrate how to specify a conduit with our approach. Finally Section 7 presents the concluding remarks and discusses future work.

2. Problem domain analysis

Domain analysis is a process by which one studies a domain to build up a thorough understanding of an entire system and investigates the ways of modelling a domain [13]. This section analyses Palm OS data synchronisation to discover a set of common operations

Proceedings of the International Conference on Mobile Business (ICMB'05)
0-7695-2367-6/05 $20.00 © 2005 IEEE

in the domain and to establish reusable assets for future system development.

2.1. Overview

On the Palm OS, a 'database' is just an array of memory chunks with no fields, no indexing and no high-level access via SQL. The reason for this simplicity is the limitation of the memory size and the primary goal of fast performance of the Data Manager [15]. As such the structures of handheld records are 'packed'. Hence, when those handheld records need to store on the desktop, they have to be 'unpacked'.

The data synchronisation of Palm OS handhelds and desktop computers is conducted by Palm OS conduits. A conduit is the program that HotSync Manager [4], a desktop application, runs to synchronise databases on the handheld with their desktop counterparts. A conduit can be created by using the Conduit Development Kit (CDK) [3] which contains all the templates, object classes, and documentation for Mac OS or Windows operating systems. To develop conduits for Mac OS, C/C++ Sync Suite [2, 12] is the only choice, while on Windows C/C++ Sync Suite, JSync Suite and COM Sync Suite [3] can be used to create conduits in C/C++, Java and Visual Basic.

Most synchronisation suites provide developers with two approaches to create conduits. That is Sync Manager API and Generic Conduit Framework (GCF) [3]. The Sync Manager API is a low-level programming interface to HotSync Manager for direct communication between a conduit and a handheld. On the other hand the GCF is a set of classes that work together to carry out functions of a conduit. It derives certain classes from base classes and customises them for a desired data format on the handheld or the desktop computer.

The types of Palm OS data synchronisation can be classified into two broad categories: one-way To- or From-Handheld and Mirror-Image synchronisation. The one-way synchronisation only allows data modifications on one side and overwrites the data on the other end, whereas the Mirror-Image synchronisation allows data modifications on either the handheld or the desktop computer and makes data identical on both sides. A record modification can be added, modified, deleted or archived. The GCF provides 'Sync Logic' [3] to handle a variety of record modification situations on both sides and ensure correct synchronisation.

2.2. Current solution

Currently, Palm OS conduits are developed with the programming languages mentioned above. As features of different languages vary, the choice of a programming language mainly depends on the developer's personal preference and individual application's requirement. The C/C++ Sync Suite provides a superset of all features that cover all requirement of a conduit development [3]. Thus this paper focuses on investigating C/C++ Sync Suite and only on Windows, since C/C++ Sync Suite on both platforms is similar and there are some limitations for Mac OS. Our work can be extended to support Mac OS platform and other suites easily.

Figure 1 generalises how the C/C++ Sync Suite implements Palm OS conduits. In general a conduit can be implemented by using GCF (path 1) or Sync-Manager API (2); and in both scenarios a single conduit can have multiple-synchronisation actions (3) or a single-synchronisation action (4). If they are multiple-synchronisation actions (3), then they divide into single-synchronisation actions (4) and then are implemented one by one. A single-synchronisation action (4) may synchronise one-to-one database (single-database synchronisation) (6) or one- (many-) to-many databases (multiple-database synchronisations) (5); and the case of multiple-database synchronisations (5) should always break down to single-database synchronisations (6) in a similar fashion to the synchronisation action. For a single-database record (6), its synchronisation type can be Mirror-Image (7) or one-way synchronisation (8), and then it is implemented accordingly.

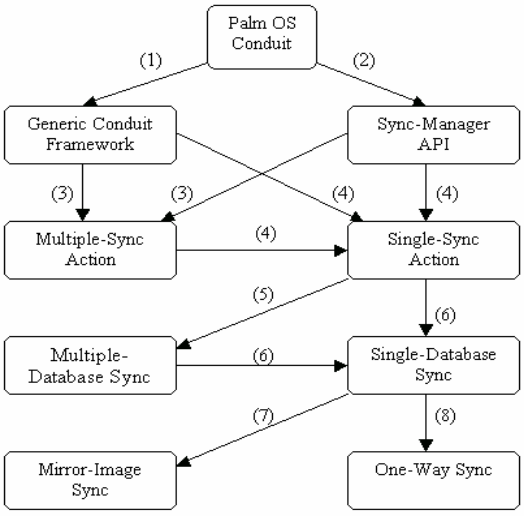

Figure 1. Taxonomy of Palm OS conduit implementation

2.3. Conduit wizard

A Palm OS conduit development with C/C++ Sync Suite (Figure 2) uses the conduit wizard [4], a type of code wizard in Microsoft Visual C++ .Net, to generate skeleton code and then let developers fill the gaps with their own code to produce a final conduit program. This method certainly pushes conduit design one level up than previous versions, but it is far too general and broad, relying on ad-hoc user supplied code to provide more specific functionality. In some cases it may produce some redundant code to perform unnecessary work during the synchronisation, such as using the GCF to generate skeleton code for a one-way synchronisation conduit. Most one-way synchronisation simply 'overwrites' from one end to the other, but the code generated by using GCF reserves all the memory space and checks record data flags in advance, which is not necessary. On the other hand, if a one-way synchronisation is expected, then the conduit wizard provides little choice and generates no useful code [1, 3, 15].

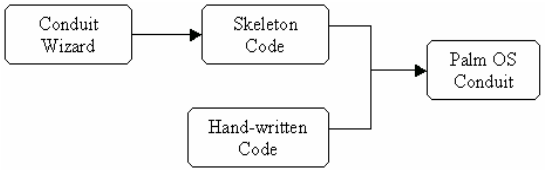

Figure 2. The process of generating Palm OS conduit in Visual C++

3. Domain problems and the solution

From our domain analysis, the disadvantages of the current conduit wizard approach are:

- **Low-level programming:** For designing a conduit, the choice of an algorithm and its implementation totally depends on the developer's domain knowledge and language development experience.
- **Error-prone cut-and-paste method:** There is a repetitive code sequence in manual conduit development. With the cut-and-paste method, it is easy to make mistakes that lead to errors at compile-time or, even worse, unpredictable behaviour at run-time.
- **Irreversible:** With the conduit wizard approach, the process of a conduit development is irreversible. Once programmers start working on the skeleton code, there is no way to modify or reset the conduit wizard and still keep the partially finished work.

The major design goal for conduits is to minimize synchronisation time. This means that the simplest method of synchronisation should be chosen for a conduit design. The overall result of the domain analysis suggests that a language is needed to provide a high-level software interface for domain developers, and EDSL techniques are suitable for describing this domain problem.

4. Embedded domain-specific languages

A domain-specific language (DSL) is a programming language that is tailored for a specific set of tasks. The key advantages are that a DSL can reduce the cost of production and increase the domain expressive power by its declarative nature [17]. The reason is that DSLs can separate domain experts from low-level details, and focus on high-level decisions for addressing domain issues.

Nevertheless, designing a DSL involves considerable work to write a compiler from scratch. This fact has led to the popularity of embedded DSLs (EDSLs). An EDSL is just a subroutine library defined as a set of functions and encapsulated in modules of an existing 'host' language. It inherits generic language constructs of its host language and adds domain-specific primitives that allow programmers to work at a higher level of abstraction; hence it is more economical to design, implement and maintain. Since functional languages, especially Haskell, have several useful features such as expressiveness, high-order functions and strong typing with polymorphism, they are particularly suitable to be host languages [5].

However there are some drawbacks of the EDSLs approach, such as syntax that is often far from optimal, poor error message reporting at the domain level and an inability to perform domain-specific optimisations [10]; but in the functional setting and in particular if Haskell is used for a host language, some of these shortcomings can be reduced by using monads [6] for domain-specific optimisation and partial evaluation [6] for overall optimisation.

5. Conduit EDSL design and implementation

Figure 3 illustrates the top-level design of Conduit EDSL. Its input is a domain specification and its output is a Palm OS conduit program. Firstly we derive an abstract syntax for our Conduit EDSL based on the domain analysis. To make the Conduit EDSL more usable, we construct a high-level interface to provide a

simple and type safe language to domain programmers. The implementation of the interface maps the code written in the Conduit EDSL down to a conduit abstract syntax tree (AST) [16]. Each node of the conduit AST corresponds to a building block or a function that generates a block of code. When all nodes within conduit AST are traversed, a complete target Palm OS conduit program is produced.

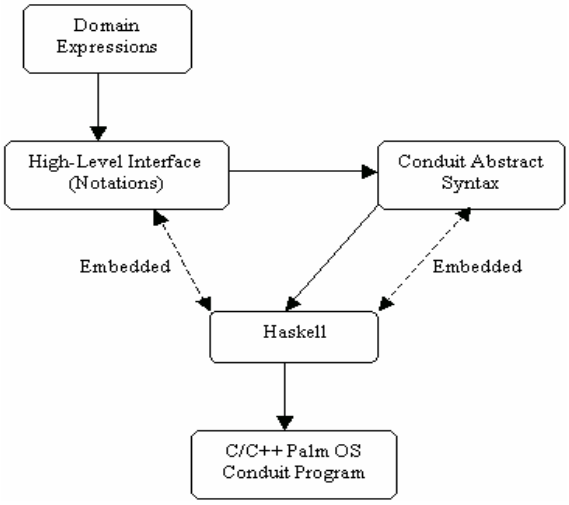

Figure 3. Simplified view of Conduit EDSL

5.1. Abstract syntax

Using the domain knowledge collected in the analysis phase and EDSL techniques discovered previously, the conduit abstract syntax could be defined and presented as Haskell datatypes to provide a firm semantic base. These datatypes are used for structuring possible conduits that domain developers are targeting and preventing the construction of syntactically incorrect specifications [11].

On the top level, the abstract syntax `Conduit` (Figure 4) is simply a list of entries and a Boolean value to indicate if a configuration dialogue ('Change HotSync Action') box is needed for this conduit to let the user change the synchronisation direction. The `Entry` definition consists of synchronisation data for both sides, a synchronisation policy to be performed and two Boolean values – here the synchronisation policy is either Mirror-Image, From-Handheld or To-Handheld.

As seen from Figure 4, *SyncData*, *HhRecord*, *PcRecord* and *Field* carry synchronisation data in different ways. At some stage defined constructors are used to match the pattern of a record, such as *Define* and *Struct*; and at some other stage a list is employed to group fields for managing them easily, for example *[[Field]]* in *PcRecord*. The main part of the conduit abstract syntax is shown in Figure 4, whereas minor parts are omitted for simplicity.

```
type Conduit       = (ChangeDlgBox, [Entry])
type ChangeDlgBox  = Bool
type Entry         = (SyncData, SyncPolicy,
                      Archive, CatgrySupport)
data SyncPolicy    = Mirror | FromHH | ToHH
type Archive       = Bool
type CatgrySupport = Bool
type PrivatSupport = Bool
type SyncData = (HhDb, PrivatSupport, PcDb)
type PcDb     = (PcDbPath, PcDbType, PcRecord)
type PcRecord = (StructName, [[Field]])
data HhDb     = AddrBook | DateBook | MemoPad
              | ToDoList | Custom HhApp
type HhApp    = (AppVers, CreatorId, HhRecord)
data HhRecord
       = Define Def Token Rep
       | Struct StructName Construct [[Field]]
type StructName = String
type Construct  = [(FieldName, Field)]
type Field  = (Ftype, FieldName)
data Ftype  = Int | Unlong | Long
            | Unshort | Short
            | Unchar | Char | Stype StructName
data FieldName = Cst String  | Lst String
               | Nls String  | Pnt String
   ...
```

Figure 4. Conduit abstract syntax

5.2. High-level interface

For making the Conduit EDSL more abstract and declarative, a conduit high-level interface is constructed to allow domain developers to focus their attention more on the essential elements that need to be specified by hiding the conduit abstract syntax. In the following introduction of the high-level interface, a running example is used to demonstrate how a conduit is specified. This simple conduit is called 'SalesConduit'. It only performs one-way synchronisation and provides the 'Change HotSync Action' dialog box for users. Suppose there is only one database 'Order' in this application. The handheld application's creatorID is 'SLES' with application version number 0. The file format of the desktop database is 'text' with a default path. Assume there is no need to hide private records, to support category synchronisation or to handle record archiving.

Suppose the record 'Order' is synchronised by copying *OrderedItem* information to the desktop

computer and customer details to the Palm handheld. The C++ language raw record data structures from the handheld application are shown below.

```
struct OrderedItem {
  long           orderID;
  unsigned long  productID;
  unsigned long  quantity;
};
struct PackedOrder{
  long           customerID;
  char           custName;
  unsigned short numItems;
  OrderedItem    items[1];
};
struct Order {
  Order(unsigned short num) {
      custName = 0; numItems = num;
      items = new OrderedItem[numItems];}
  ~Order(){delete [] custName;
           delete [] items;};
  long            customerID;
  char           *custName;
  unsigned short  numItems;
  OrderedItem    *items;
};
```

In order to make the interface easier to understand and to construct, two more datatypes are introduced. They are *FeaturesAndDBInfo* and *ExtraSyncData*.

FeaturesAndDBInfo simply groups desired conduit features and related database details. Three Boolean values indicate if these features are supported in this conduit.

```
type FeaturesAndDBInfo
  = (AppVers, CreatorId, PcDbPath, PcDbType
    ,PrivateSupport, Archive, CategorySupport)
```

According to the requirement of 'SalesConduit', there is only one handheld application involved and its version number is 0, so it is a new application and specified as '*NewAp 0*'. The rest of specifications are straightforward.

```
ft_db = ( NewAp 0, "SLES", Deft, Txt
        , False, False, False)
```

If a record member is user-defined, its structure can be defined as a C++ 'struct' definition or a 'typedef' declaration. For specifying the maximum items of this member within single record or the array size in the typedef declaration, we use the constructor *Sdefine Int* to assign this number.

```
data ExtraSyncData = Xstruct [Field]
                   | Tdefine Int Ftype
                   | Sdefine Int
```

In the 'SalesConduit' example, *OrderedItem items* is the user-defined field member within the record 'Order'. The members of 'struct' *OrderedItem* can be specified through the datatype *ExtraSyncData*. That is:

```
ordItem = Xstruct [slng "orderID", sulg "productID"
                  ,sulg "quantity"]
```

In this specification, the functions *slng* and *sulg* are unknown. But from their structures and the constructor *Xstruct*, we can guess that the two combinators are used to construct fields, which will be introduced next.

First of all, a group of combinators is defined with the same type signature for composing a single field from a given field name.

```
sulg :: String -> Field
slng :: String -> Field
scha :: String -> Field
...
```

By using these combinators, field types are assigned accordingly. The example of the constructor *Xstruct* in *ordItem* shows how to use these combinators to specify fields. In that example, *slng "orderID"*, *sulg "productID"* and *sulg "quantity"* correspond to the field specifications with the conduit datatypes: *(Long, Cst "orderID")*, *(Unlong, Cst "productID")* and *(Unlong, Cst "quantity")* respectively.

From the domain analysis, it is known that fields on both ends share the same name and type normally. With this fact, there is no need to enter fields separately for both sides; instead one field name is required for constructing fields for both sides. The field types of the fields generated below are in turn *long*, *unsigned short* and user-defined type.

```
lng :: String -> (Field,[Field])
ush :: String -> (Field,[Field])
stp :: String -> (Field,[Field])
...
```

For instance, if we want to specify the fields '*customerID*', '*numItems*' and '*items*' for both sides in the 'SalesConduit' example, then they look like:

```
lng "customerID"                and
ush "numItem"                   and
stp "OrderedItem"
```

which correspond to the field specifications with the conduit datatypes:

```
((Long, Cst "customerID"), [(Long, Cst "customerID")])

((Unshort, Cst "numItem"), [(Unshort, Cst "numItem")])

( (Stype "OrderedItem", Lst "orderedItems")
, [(Stype "OrderedItem", Pnt "orderedItems")])
```

The reason for using a list to manage the second element of the tuple is to cooperate with the combinator *cha*, which takes a field name and a list of field names. It returns one field from the first argument for the handheld side and the list of fields by combining first and second arguments for the desktop part. The second

argument can be an empty list if there is only one field on the desktop corresponding to the one on the handheld.

```
cha :: String -> [String] -> (Field, [Field])
```

The fields of the customer name in 'SalesConduit' can be specified through `cha`:

```
cha "custName" []
```

which correspond to the field specifications with the conduit datatype:

```
((Char, Lst "custName")
,[(Pnt, Cst "custName")])
```

Once fields for both sides are structured, it is easy to specify a record, which just groups related fields. We create three operators (`><`, `<|` and `|>`) to combine fields together with relevant information to perform Mirror-Image, From-Handheld and To-Handheld synchronisation, respectively. In these operators, the types `a`, `b` and `c` stand for *(Field, [Field])*, *[HhRecord]* and *Construct* respectively.

```
(><) ::[a] -> ([a],b,c) -> ([a],b,c,SyncPolicy)
(<|) ::[a] -> ([a],b,c) -> ([a],b,c,SyncPolicy)
(|>) ::[a] -> ([a],b,c) -> ([a],b,c,SyncPolicy)
```

Note that the second parameter has a type of *([a], b, c)*. This is a special case where a field is user-defined. The lower fixity operator (`<:`) is used to specify such a field with a defined structure and its dependent data information. It takes a tuple with a pair of *(Field, [Field])* and a list of extra data.

```
(<:) :: ((Field, [Field]), (Field, [Field]))
   -> [ExtraSyncData]
   -> ([(Field,[Field])],[HhRecord],Construct)
```

The second element of the tuple is a user-defined field with its definition or structure specified in the list of extra data, while the first element is a field to support the second element construction. For example, the second element is a structure definition and members of this structure are given in the extra data list, while the first element is the field for defining the size of the structure. In return, the two elements within the tuple combine into a list, the structure of the user-defined field and its supplementary statements are reconstructed as a list of *HhRecord* for referencing, and some elements of the record constructor in the target code are obtained from the first argument.

By using the operators introduced above and the raw record data structure supplied in the beginning of this section, we can specify these two field groups as:

```
custDetail = [lng "customerID", cha "custName" []
            ] |> ([],[],[])
ordItems = [] <| ( ush "numItem"
                 , stp "OrderedItem"
                 ) <: [ordItem]
```

On the top level, the function *entry* is defined by taking a record name, conduit features, database details and a list of record detail (which is constructed by the combinators defined above with generic type *([a], b, c, SyncPolicy))*.

```
entry :: String -> FeaturesAndDBInfo
      -> [([(Field, [Field])], [HhRecord]
           ,Construct, SyncPolicy)]  -> [Entry]
```

Normally one record entry only performs one synchronisation policy, which is one *Entry* in the conduit abstract syntax. But the function `entry` returns a list of *Entry*. The reason is that with defined combinators and extra datatypes, it is possible to specify an entry with multiple synchronisation actions and synchronise the data on a field level. When the function *entry* is executed, each synchronisation action with related fields is mapped to single conduit *Entry*. So the 'Order' record specification entry now looks like:

```
ent = entry "Order" ft_db [custDetail, ordItems]
```

5.3. Code generation – the AST traversal

Code generation is the process of transforming a high-level specification to a related low-level target code. As such, the conduit code generation is really to build a target conduit by traversing each node of a conduit AST and composing related building blocks. For the node that contains sub-nodes, a further traverse is required. This section only shows a few steps of the implementation to demonstrate the strategy used for the conduit generation.

On the top of a conduit AST, there is only one parent node representing a desired conduit. The function *creatCond* generates a complete conduit target code by taking a conduit specification and a related handheld application's name, which is used for naming the conduit.

```
creatCond :: String -> Conduit -> IO()
creatCond appName (changeBox, entrs) =
  do ...
     p1 <- readFile (defDir ++ "GenCond2.cpp")
     writeFile "Tmp2.cpp" p1
     mapM_ singleEntry2c entrs
     ...
```

In this function, the building blocks can be mounted together in a number of ways according to the individual conduit AST. Note that the constant code blocks are stored in a default directory (*defDir*), but this can be modified to let the programmer specify a directory. Temporary files are used to hold building blocks for major construction, while some codes are filled between the building blocks, which are variations among the conduit family. Since a conduit may contain

multiple entries, the function *singelEntry2c* is employed to map each conduit entry to the related code depending on its synchronisation policy.

```
singleEntry2c :: Entry -> IO()
singleEntry2c
       (syncData,syncPolicy,archiv,catgry)
 = case syncPolicy of
      FromHH -> fromHh syncData archiv catgry
      ToHH   -> toHh syncData archiv catgry
      Mirror -> mirror syncData archiv catgry
```

The sub-node *fromHh* includes synchronisation data and the features of this entry. With this information, the related code blocks can be generated and appended to the temporary files. Note that the function `render` (a built-in function of the module *PPrint*) is used to convert the *Doc* (the type of the function *enpFromHh*) to *IO* for appending.

```
fromHh :: SyncData -> Archive
       -> CategorySupport -> IO()
fromHh (hhDb, privatSupport,pcDb) ar ct
 = do appendFile "Tmp1.cpp"
         (render (enpFromHh hhDb pcDb ar ct))
      ...
```

Following the conduit AST from the top-down, the code generator traces down a list of records one by one and in a similar fashion for fields. When visiting down to the bottom of a conduit AST, field structures can be expressed with related target code according to specifications. The function *field2c* simply combines the function *ftype2c* and *fname2c*, which then just go through their possibilities and transform into the target code accordingly.

```
field2c :: Field -> Doc
field2c (ftype, fname)
 = case fname of
     Nls _ -> text "new" <+> ftype2c ftype
                         <> fname2c fname
     otherwise -> ftype2c ftype
                  <+> fname2c fname
```

6. Conduit generation example

To view the whole picture of the Conduit EDSL, the example in Section 5.2 will be revisited. With the specified the record 'Order', the developer can then write the `main` function (Figure 5) within the Conduit EDSL. As expected, the type of the *main* function is *IO()*, since the generated code is printed out to target files. A directory for storing generated files can be specified explicitly by using the function *createDirectory* and *setCurrentDirectory* to create and set it as the current. Both functions are from the built-in module *Directory*, which handles the file system and directory management.

```
import CreatConduit
import Embedding
import Directory

main :: IO()
main =
 do createDirectory fp
    setCurrentDirectory fp
    creatCond "Sales" (True,ent)
 where
  fp = "C:/MyDir/MyConduit"
  ent = entry "Order" ft_db [custDetail,ordItems]
```

Figure 5. Palm OS conduit generation example

Once required modules are imported and the directory is created and set, the conduit high-level interface can be accessed by using the function *creatCond*. It takes the handheld application's name 'Sales', and the conduit entry details which are specified in Section 5.2. When this function is executed, it automatically generates the target Palm OS conduit program.

In the simple case like the one in Figure 5, two supporting files and two C++ files *SalesGenCond.h* and *SalesGenCond.cpp* are produced. The supporting files are standard so they are just copied across. The C++ files employ the usual convention that the '*.h*' file gives the representation and some function definitions, and the '*.cpp*' file contains the function implementations and the remaining function definitions. These two files contain about eight hundred lines of C++ code. In contrast the developer only need write 20 lines of code in the Conduit EDSL.

However, the code generated from the Conduit EDSL is far from concise or simple. This is because the code generation strategy used is focused on iterating over the collection of conduit entries. These entries do have a logic connection on the top level. But when executed, they are broken into separate entries and dealt one by one with no link to each other. Nevertheless, a conduit specified in the Conduit EDSL, like the one in Figure 5, is much higher in expressiveness than the one written in a C++ that is similar to the generated code. Intuitively the latter seems to be more problematic.

7. Conclusion and future works

In this paper we have introduced the use of EDSL techniques to solve Palm OS data synchronisation problems. We analysed the problem domain to collect the domain knowledge and defined the conduit abstract syntax as Haskell datatypes to provide a firm semantic base. As such, the complexity of a conduit development is reduced and tedious low-level details are hidden. For making the specification task easier, we then build a high-level interface on top of the conduit abstract syntax by

focusing on the conduit behaviour rather than its structure, which increases the expressive power and allows the domain developers to specify conduits on the field-level. Finally, we demonstrated how to turn embedded expressions into target code automatically to increase productivity based on the Prettyprinting techniques.

Conduit EDSL is still in development. This paper is only a first step in the direction of using EDSL techniques to solve Palm OS data synchronisation problems. Currently the basic implementation of one-way and Mirror-Image data synchronisation with the EDSLs approach is complete. However, it was noticed that the size of generated code may be slightly bigger than hand-written code and repetitive information is inevitable in this approach. The question raised by this paper regards how well EDSL techniques are applied to address domain problems and to achieve high-level abstraction, expressiveness and productivity by not compromising efficiency and correctness. Two main directions in which this work may be extended are envisaged.

- **Adjust the scope of the problem domain:** As declared in Section 5.2, the Conduit EDSL could support data synchronisation on a field level conceptually. But this argument seems weak, because the assumption is that such separated databases exist prior to the conduit development. It just shifts part of the low-level detail from the current domain to another domain. If the scope of the problem domain is increased to include database creation, we could then generate the code for both database creation and its data synchronisation, which entirely isolates domain programmers from the low-level detail.
- **Supporting domain-specific reasoning:** Functional languages such as Haskell provide us with powerful proof techniques [7]. Furthermore, domain notations are embedded into Haskell, which captures the domain semantics concisely. So formal reasoning can be directly proved within the domain semantics rather than within the semantics of the programming language [5], which ensures that the correctness of the Conduit EDSL implementation and code generation.

8. References:

[1] Foster, L.R., *Palm OS: Programming Bible*. 2nd ed: IDG Books Worldwide, Inc. 2002.

[2] Gossett, B., *C/C++ Sync Suite Companion*, in *Palm OS Conduit Development Kit for Windows, Version 4.03*. PalmSource, Inc. 2002.

[3] Gossett, B., *Introduction to Conduit Development*, in *Palm OS Conduit Development Kit for Windows, Version 4.03*. PalmSource, Inc. 2002.

[4] Gossett, B., *Conduit Development Utilities Guide*, in *Palm OS Conduit Development Kit for Windows, Version 4.03*. PalmSource, Inc. 2002.

[5] Hudak, P., *Building Domain-Specific Embedded Languages.* ACM Computing Surveys, **28**(4es). 1996.

[6] Hudak, P. *Modular Domain Specific Languages and Tools*. In *Proceedings of the Fifth International Conference on Software Reuse (JCSR '98)*. p. 134-142. IEEE Computer Society. 1998.

[7] Hudak, P., *The Haskell School of Expression: Learning Functional Programming Through Multimedia*: Cambridge University Press. 2000.

[8] Hughes, J. *The Design of a Pretty-Printing Library*. In *Proceedings of First International School on Advanced Functional Programming*. p. 53-96. LNCS 925, Springer-Verlag. 1995.

[9] Jones, S.P., *Haskell 98 Language and Libraries*: Cambridge University Press. 2003.

[10] Kamin, S.N. *Research on Domain-specific Embedded Languages and Program Generators*. In *Proceedings of Theoretical Computer Science*. Elsevier Science B. V. 1998.

[11] Leijen, D. and Meijer, E. *Domain Specific Embedded Compilers*. In *Proceedings of 2nd USENIX Conference on Domain-Specific Languages (DSL)*. p. 109-122, Austin, USA. 1999.

[12] Loy, G. and Gossett, B., *C++ Generic Conduit Developer's Guide*, in *Palm OS Conduit Development Kit for Windows, Version 4.03*. PalmSource, Inc. 2002.

[13] Prieto-Diaz, R., *Domain Analysis: an Introduction.* ACM SIGSOFT Software Engineering Notes, **15**(2): p. 47-54. 1990.

[14] Reveillere, L., et al. *A DSL Approach to Improve Productivity and Safety in Device Drivers Development*. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*. p. 101, Grenoble, France. 2000.

[15] Rhodes, N. and McKeehan, J., *Palm OS Programming: The Developer's Guide*. 2nd ed: O'Reilly & Associates, Inc. 2002.

[16] Scott, M.L., *Programming Language Pragmatics*: Morgan Kaufmann. 2000.

[17] van Deursen, A., Klint, P., and Visser, J., *Domain-Specific Languages: An Annotated Bibliography.* ACM SIGPLAN Notices, **35**(6): p. 26-36. 2000.