The Architectural Design of FRUIT: A Family of Retargetable User Interface Tools

Yi Liu
Computer Science
University of Mississippi
University, MS 38677

H. Conrad Cunningham
Computer Science
University of Mississippi
University, MS 38677

Hui Xiong
Computer Science
University of Mississippi
University, MS 38677

Abstract

Forms-based user interfaces are widely used means for human-computer interaction. However, it is difficult to exploit software reuse in forms-based systems because the systems tend to be dependent upon the particular display technologies used. This paper introduces the Family of Retargetable User Interface Tools (FRUIT) that aims to solve this problem. FRUIT adapts the device-independent XForms technology to deliver a family of products capable of supporting a variety of display technologies. This paper presents the architecture for the FRUIT product line and illustrates the design methodology used—a methodology based on stepwise refinement and commonality and variability analysis; it also describes two members of the FRUIT family and shows how their designs and implementations are refinements of the family architecture.

Keywords: FRUIT, product line, user interface.

1. Introduction

Forms-based user interfaces are widely used means for human-computer interaction. Providing users with a familiar data processing style that is similar to the paper equivalent, a forms-based user interface is easy to understand and friendly to use. It can exist on many kinds of devices and may be implemented by various programming languages. Sometimes a software provider needs to implement several user interface versions for one application since users might have different preferences or means for running the software. For example, some may access the software through a Web browser, while others may use a standalone version. These products belong to a software family.

Figure 1 shows a general architectural design of an application. In this architecture, the user interacts with the application through a forms-based User Interface (UI). Ideally, the implementation of the application components should be independent of the UI component; thus, for developing a new product in the software family with a different user interface, the software provider only needs to develop a new UI component corresponding to the new device and plug it in. However, implementations of user interfaces on different devices are usually not the same; for example, an HTML-based user interface and a Java Swing-based user interface require considerably different implementations. Therefore, the variety of devices makes it difficult for several family members to share code or fine-grained architectural design within a forms-based UI component.

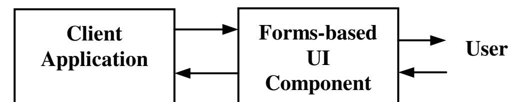

Figure 1. An architectural design of an application

In the real world, the implementation of the *Client Application* often cannot be completely device independent. For example, the first member of the family might have a JavaServer Pages (JSP) UI component with the application components implemented in Java. But later, a new family member may be added that demands a Perl/Tk UI component. The existing application components are not compatible with the UI component. So, based on this architectural design, even though the members of the family share the same application functionality, it is difficult for them to exploit software reuse.

The Family of Retargetable User Interface Tools (FRUIT) aims to solve these problems. FRUIT is a software product line under development in which the members might be implemented in various ways—by Java Swing, by JSP plus HTML in a browser, by Perl/Tk, and so on. They all share a high-level architectural design, a common interface design and perhaps some components. By using FRUIT, the form of and interactions with user interface controls can be represented in a device-independent manner and translated into specific user interface technologies.

Figure 2 shows an architectural design of an application with FRUIT. The information exchanges between the *Client Application* and FRUIT use the XML-based XForms standard [11]. FRUIT takes two XML documents from the *Client Application*—the form configuration information and form description—that describe how a form is expected to look. It then processes the form and presents the form to the user. Because of FRUIT, the application components are totally independent of the user interface.

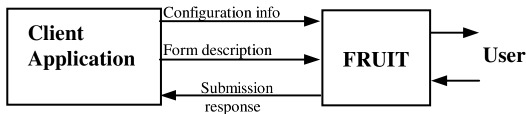

Figure 2. An application with FRUIT

This paper is organized as follows. Section 2 briefly presents the features of the XForms technology. Section 3 describes the architecture of FRUIT in detail. Section 4 uses two family member candidates—Java Swing and Java Servlet systems—to illustrate the differences in implementation among family members. Section 5 concludes the paper and suggests possible future work.

2. XForms

The means for exchanging information between FRUIT and the client application (Figure 1) is XForms. XForms is a new device-independent markup language that seeks to combine XML and forms [11]. Unlike traditional HTML forms, XForms separates the data being collected from the form presentation. XForms splits traditional HTML (or XHTML) forms into three parts: the *XForms model* that describes what a form does, the *XForms user interface* that describes how the form is to be presented, and the *instance data* that represents the values and state of all the instance data nodes associated with the form [11]. Figure 3 [11] shows the three parts of the XForms model.

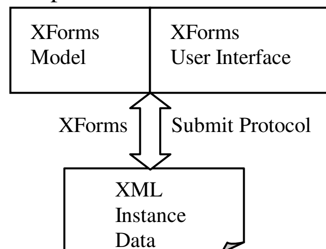

Figure 3. Three parts of XForms

Consider a simple form example: a form presents a textbox to the user and the user fills it in and then submits the form. Figure 4 shows the layout of the form. Figure 5 gives the XForms model section. It starts with the tag `<XForms:model>` (or just `<model>`) and ends with the tag `</XForms:model>` (or `</model>`). Normally this is contained within the `<head>` section of an XHTML document. The XForms model for this example says that one piece of information (`name`) is to be first collected and then submitted via the URL given in the `action` attribute. In this example, the initial value of the instance data is set to be empty within the XForms model in the `<head>` section.

Figure 4. Layout of an example form

```
<head>
   <model>
     <instance>
       <example>
           <name/>
       </example>
     </instance>
    <submission action =
        "http://cs.olemiss.edu/submit"
        method ="post" id = "submit"/>
     </model>
</head>
```

Figure 5. XForms model and instance data

The user interface specification is shown in Figure 6. This figure specifies the items that are defined in the XForms model (Figure 5) between the starting tag `<XForms:instance>` (or just `<instance>`) and the ending tag `</XForms:instance>` (or `</instance>`). It describes the form layout that normally appears within the `<body>` section of an XHTML document. XForms defines "a device-neutral, platform-independent set of form controls" [11], such as input textboxes, checkboxes and submission buttons, for general purpose use. In this example, the form control `input` is used to show a textbox and the control `submit` is used to show a submission button.

```
<body>
       <input ref = "name">
           <label> Your Name </label>
       </input>
       <submit submission = "submit">
         <label> submit </label>
       </submit>
</body>
```

Figure 6. User interface section

After the user fills in the form (as shown in Figure 7) and clicks the `submit` button, the submitted instance data is serialized as an XML document (shown in Figure 8) by the XForms processor.

Figure 7. Filled form

```
<example>
        <name>yi liu </name>
</example>
```

Figure 8. Submitted instance data

As shown in this example, the representation of a form and the collection of its data are obviously separated. It is the separation of presentation from content that allows reuse and offers device–independence [7].

XForms is a general description for user interface controls. It does not directly deliver the form layout to the user on the device. There must be a processor that renders the form description and presents it to the user. Currently, XForms does not have widespread Web browser support. XForms implementation products, such as XFE [6], which is an XML forms engine, and FormsPlayer [12], interpret XForms controls in the Web browser and process the user interactions.

Most of the XForms products use a Web browser as the medium for the user interface layout. However, the required user interface does not always use a browser. It could use some other medium, such as a Java Swing application. FRUIT is designed to make it easy to change from one medium to another.

XForms provides a device–independent forms control description. A FRUIT processor takes an XForms description document as input, interprets it and presents the user with form controls appropriate for the chosen user interface device. The FRUIT processor then collects the user responses and returns them as XML documents. FRUIT is designed to be a family of processors sharing a common architecture. The architecture is designed to be flexible, so that family members can be implemented for a wide array of different devices and user interface technologies.

3. Architectural design

The general goal of the FRUIT project is to develop a product line by which the form of and interactions with the user interface controls can be represented in a device-independent manner and then translated into specific user interface technologies.

A software product line is "a collection of systems sharing a managed set of features constructed from a common set of core software assets." [2] These assets include a common software architecture shared by the products and a set of reusable software components [8] [10]. Since the software architecture sits above the other assets and is shared by all the members in the product line, it is the key issue in the analysis of the architecture of a product line.

3.1. Commonality and variability analysis

Commonality analysis [4] is a process used to find the common characteristics among family members. "Commonalities are a list of assumptions that are true for all the members of the family." [1] The commonalities enable designers to determine the scope of the product line's software architecture.

Variabilities are a list of assumptions that are true for only some members of the family. For one aspect, at least two members have different descriptions. Family members differ from each other in how these variabilities are realized. When we design a common architecture for a family, the variabilities are the issues that need to be hidden inside components. We usually generalize these variabilities and wrap them with abstract interfaces [3]. Thus, all the family members can still be built with the same architecture.

3.2 Stepwise refinement

In the design of FRUIT, we use a stepwise refinement [5] method that combines aspects of domain (requirement, actually) analysis and commonality and variability analysis. Here we describe three steps of the refinement.

Step 1: ***Find the main components from the general description of FRUIT and specify the top-level architecture.***

In the chosen design, FRUIT takes descriptions from the *Client Application*, translates them into a form (*User Interface*) and delivers the form to the device. Then, FRUIT collects data from the *User Interface* and sends the data back to the *Client Application.*

Based on the description of the FRUIT, the main components that directly interact with FRUIT are the *Client Application* and the UI. The initial architecture can be designed as shown in Figure 9.

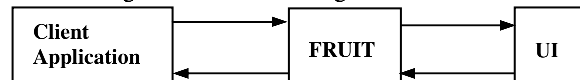

Figure 9. FRUIT architecture – step 1

Step 2: ***Analyze the interfaces among the main components.***

Commonality analysis reveals the need for two interfaces in the family's architecture.

Commonality a: FRUIT is designed and implemented independently from the *Client Application*. Because of the need to maintain this separation for all the members of the FRUIT family, we assume that all communication between the *Client*

Application and FRUIT uses the abstract interface between them.

For each FRUIT family member, the inputs consist of both the *form description* and the *form configuration information*. As described in the section 2, to gain the benefit of device independence we choose to implement the *form description* as an XForms document. This document describes the elements that a form contains. The form description is required for each family member. The *form configuration information* is an XML document. It specifies details, such as the layout of a form, expected control for an element, and so on, but is not mandatory for members. If a member doesn't have any specific information on the form configuration, then the *form configuration information* is an empty document. The data submitted from the UI are collected by FRUIT and sent back to the *Client Application*.

Commonality b: The UI may vary from one device to another. To establish this variability factor from the architecture design, we assume that each FRUIT member has an interface between FRUIT and UI that isolates FRUIT from the specific device characteristics.

The interactions between FRUIT and UI are as follows: FRUIT sends a form to the device and then collects the data that user inputs using the UI. Figure 10 shows the architecture after adding the interactions described by *Commonalities a* and *b*.

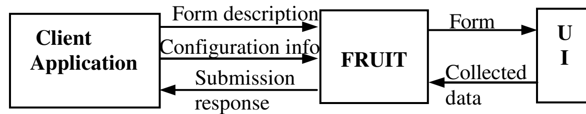

Figure 10. FRUIT architecture – step 2

Step 3: *Refine the main components.*

After we analyze the relations among the main components, we need to specify each component. Since the *Client Application* is supplied by clients and the UI is created by FRUIT, FRUIT is the only component that needs to be refined.

The parts of FRUIT that take the inputs from the *Client Application* might be divided into two stages. At the first stage, all FRUIT needs to do is to parse the XForms and then generate the layout and deliver the form to the chosen device.

Step 3.1: *During the first stage, the commonalities can be described as follows.*

Commonality c: Each family member needs to parse the XForms documents. This is the first step for processing the input documents. As discussed in section 2, XForms are not supported by every kind of device. Thus, FRUIT needs to parse the XForms and extract the elements. This subcomponent can be reused by those family members that are designed to use the same programming language. For example, a Java Swing application and a Java Servlet application may share the same *XForms Parsing* component. In a Java prototype for FRUIT, JDOM [9] can be used for this purpose. By adding *Commonality c* into the FRUIT, the architecture becomes that shown in Figure 11.

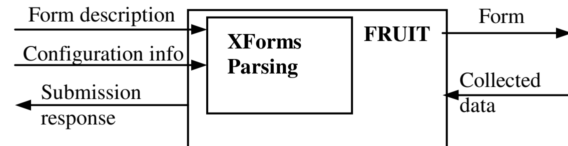

Figure 11. FRUIT architecture – step 3.1

Commonality d: Each family member generates the displayable forms. A form is generated based on the *form description* and the *form configuration information*. FRUIT parses the *form description* and extracts the elements to be displayed. Each element is translated into a displayable control that satisfies the expected device. If the *form configuration* is not empty, it gives the layout information for these controls. The controls assembled in certain layout construct the form. The component that achieves this functionality is *Form Generation*. By adding *Commonality d*, the architecture becomes that shown in Figure 12.

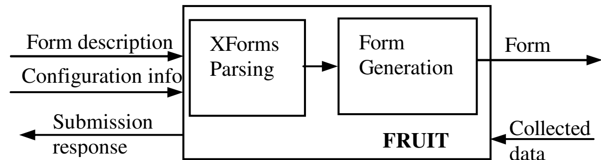

Figure 12. FRUIT architecture – step 3.1

After the *User Interface* (UI) is displayed, the user might enter data and submit the form. All FRUIT needs to do is collect the data that is submitted by the user, assemble them into an XML document and return it to the *Client Application*. This is the second stage.

Step 3.2 *At the second stage, the commonalities can be described as follows*.

Commonality e: We assume that, for all the members, the UI is the only way that a user can submit data and that the data from the user is sent back to FRUIT. We also assume that all the members use FRUIT to submit the data from the user to the *Client Application*. Based on these assumptions, all members of the FRUIT family need to collect data from the UI and send the result back to the *Client Application*.

Although the collecting processes vary from one user interface technology to another, the process that assembles the submitted data is always the same.

The submitted data contains two parts. One is the *instance data*; an example of the *instance data* is shown in Figure 8. Another is an *action description*, which indicates the next form to be invoked. For example, suppose *form1* needs to invoke *form2* after *form1*'s job is done. When *form1* returns, it should describe the URI for *form2* in the *action description* as a part of submitted data. These two parts are assembled by FRUIT into an XML document.

By adding *Commonality e* into FRUIT, the architecture becomes what is shown in Figure 13.

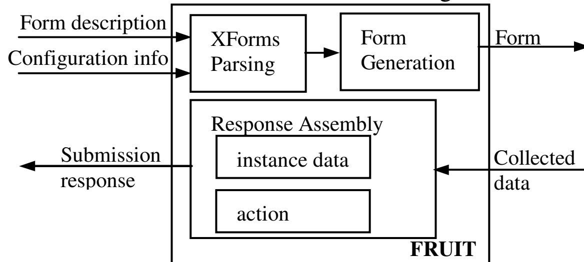

Figure 13. FRUIT architecture – step 3.2

Step 4: *Refine the subcomponents.*

The structure of component *XForms Parsing* is straightforward. Components *Form Generation* and *Response Assembly* can be refined further.

FRUIT generates a form based on the *form description* and the *form configuration information*. The *form description* does not contain any layout information. For example, as shown in Figure 6, `name` is described as an `input` control in XForms. There is no information showing the size of the input textbox. This kind of information is specified in the *form configuration information* document. For example, the `name` control is a textbox with size of 20 characters, as shown in Figure 14.

```
<name>
    <size> 20 </size>
</name>
```

Figure 14. Form configuration information

For the component *Form Generation*, the commonalities can be described as follows.

Commonality f: We assume that each element that is extracted from the *form* description is to be displayed in the UI. So, each element that corresponds to an XForms control is translated into a control in the output form. For example, `name` shown in Figure 6 is translated into `textfield` in a Swing application. Every XForms control can be translated into a control for the chosen device. This process is required for all the family members. We design component *Control Construction* to handle the building of the controls. However, for different devices, the translation from the XForms controls to the controls in the UI is usually different. Thus, we provide a general interface to the *Control Construction* component.

Commonality g: We assume that the *configuration information* is not mandatory. So the *form configuration information* document might be incomplete or even empty. In this case, FRUIT needs to establish appropriate policies for the form layout. So we specify that the component *Layout Policy* is a part of FRUIT that is shared among the family members. It takes the controls that are built in the *Control Construction* component and lays out the form. *Layout Policy* might be a knowledge-based component with complicated rules. Since this paper does not seek to present a full implementation of these components, they are not illustrated in detail.

Commonality h: We assume that each family member assembles the return data after it gets the data from the UI. However, for different devices, the data from the UI might be in different formats. To eliminate this variability in the architecture, we divide *Response Assembly* into two subcomponents: component *Data Extraction* for unifying the incoming data into the same format for all the family members and component *Data Assembly* for assembling the unified data into submitted data. *Data Extraction* has an abstract interface that takes incoming data and returns formatted data. *Data Assembly* assembles the data formatted by *Data Extraction* into submitted data. Thus, *Data Assembly* is the same for all family members.

We add *Commonalities f, g* and *h* into the architecture. After following the four-step refinement approach, the general architecture of FRUIT is built as shown in Figure 15.

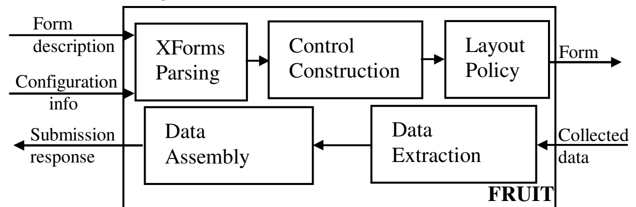

Figure 15. FRUIT architecture – step 4

4. Examples of FRUIT members

Although all members of FRUIT share the common architecture shown in Figure 15, the different technologies used by different family members lead to

different implementations. This section seeks to present the variabilities in implementations among the different members of the FRUIT product line. Two members are used as examples: one is a Java Swing application and the other is a Java Servlet application.

4.1 Variability analysis

The following variability analysis focuses on the most important differences between the Java Swing and the Java Servlet applications. In the real members' implementations, variability issues might go beyond what are described in the following.

Variability a: The action types between the FRUIT and the other two components differ in the two different implementations.

Consider *Variability a* in the FRUIT Java Swing application. This application is designed for a standalone computer. The interactions among components are procedure calls. A *Client Application* component, say C1, invokes the FRUIT processor via a procedure call. After invoking FRUIT, C1 waits for the response from FRUIT. A similar action occurs in the interaction between FRUIT and UI. Figure 16 shows the actions among the components.

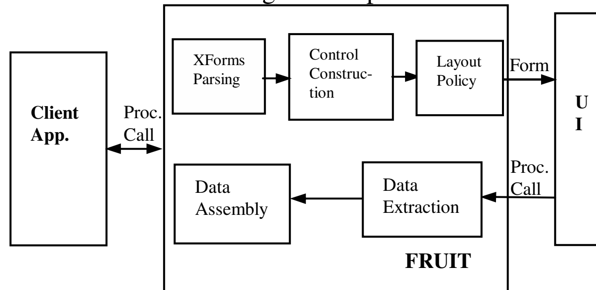

Figure 16. FRUIT Java Swing application

Now consider *Variability a* in the FRUIT Java Servlet application. This application is designed to provide a user interface in a Web browser. Differing from the Swing application whose components are communicating through procedure calls, the components of the Servlet application communicate by invoking servlets. A servlet has the property that it terminates after invoking another servlet. For example, after FRUIT generates the HTML UI and delivers it to the user, the servlet terminates. When the UI form is submitted, the servlet that is invoked to assemble the submitted data is not the previous one. Thus, we need to refine the architecture for the Servlet application.

We can divide the FRUIT process into two parts: the outgoing process, which parses the XForms and generates the form, and the incoming process, which collects the submitted data and assembles them into an XForms instance. Corresponding to these two processes, FRUIT is implemented as two servlets: one for outgoing (FRUIT-out) and another for incoming (FRUIT-in). We assume the application components are designed as servlets as well. C2's URI is embedded in C1's information when C1 invokes FRUIT. When FRUIT-out delivers the form to the UI, UI needs to hold C2's URI because the latter is what FRUIT-in needs to invoke. UI passes C2's URI to FRUIT-in. Thus, when FRUIT-in assembles the return data, in the `action` part, C2's URI should be specified. Figure 17 shows the architecture of the Servlet version of FRUIT.

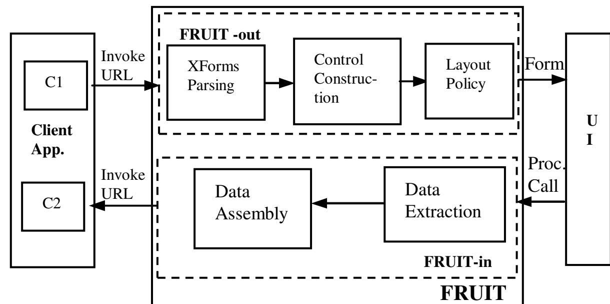

Figure 17. FRUIT Java Servlet application

Variability b: The implementation of the *Control Construction* component differs between the two applications.

Consider *Variability b* in the FRUIT Java Swing application. In this application, the extracted XForms controls are converted into Swing controls. For example, XForms control `input` is equivalent to a `JTextField` in Java Swing, and XForms control `select1` is equivalent to a `JList` in Java Swing.

Now consider *Variability b* in the FRUIT Java Servlet application. In this application, the extracted XForms controls are converted into HTML controls. For example, XForms control `select1` is equivalent to a list box in the HTML output file.

4.2 Component reuse

The Java Swing and Java Servlet applications are both implemented using Java. *XForms Parsing* and *Data Assembly* are components that can be reused for these two applications. In the *XForms Parsing* component, the JDOM library is used to parse the Form description into separate elements. The component also identifies the controls in the separated elements. The process of packaging the instance data and action into an XML document is the same for both

applications, so they can share the *Data Assembly* component.

5. Conclusion

It is difficult to exploit software reuse in forms-based systems because the systems are heavily dependent upon the devices and the display technologies used.

This paper introduces a family of tools named FRUIT that seeks to remedy this problem. FRUIT uses the new technology called XForms because XForms is designed to be independent of the devices. Although some products related to XForms exist, these products are only designed for one specific device, such as a Web browser. However, if products just focus on one special device, the independence property of the XForms is lost. To satisfy the different expectations of the users, we design FRUIT as a family of programs, or software product line, whose members share the same design architecture but differ in the implementations.

In addition to introducing a tool, this paper illustrates a methodology for designing product line architectures. This method is based on the stepwise refinement approach and commonality and variability analysis. The experience of building the Java Swing and Java Servlet prototypes of FRUIT indicates that this method results in a reliable architecture.

More work needs to be done on the design of several of the subcomponents, such as the *Layout Policy* component, which has not been extensively explored. Both of the applications implemented generate a form based on the configuration information if it exists. If the configuration information is empty, the prototypes use a sequential layout with a fixed length for each control. This strategy is not suitable for a form with controls too long to fit on one page. *Layout Policy* is an important component that should be investigated further.

6. Acknowledgements

This work was supported, in part, by a grant from Acxiom Corporation titled "The Acxiom Laboratory for Software Architecture and Component Engineering." The authors thank Cuihua Zhang for reading the paper and suggesting several improvements.

7. References

[1] M. Ardis, N. Daley, D. Hoffman, H. Siy, and D. Weiss. "Software Product Lines: A Case Study," *Software–Practice and Experience*, Vol. 30, pp. 825-847, 2000.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*, Addison Wesley, 1998.

[3] K. Britton, R. Parker, and D. Parnas. "A Procedure for Designing Abstract Interfaces for Device Interface Modules," *Proceedings of the Fifth International Conference on Software Engineering*, pp. 195-204, March 1981.

[4] J. Coplien, D. Hoffman, and D. Weiss. "Commonality and Variability in Software Engineering," *IEEE Software*, Vol. 15, No. 6, pp. 37-45, November 1998.

[5] E. W. Dijkstra. "Structured Programming," In J. N. Buxton and B. Randel, editors, *Software Engineering Techniques*, Brussels, Belgium: NATO Scientific Affairs Division, pp 84-87, 1970.

[6] http://www.e-xmlmedia.com/prod/xfe.htm, last accessed: 2 June 2003, last updated: unknown.

[7] M. Dubinko. *XForms Essentials*, O'Reilly, 2003.

[8] G. C. Gannod and R. R. Lutz. "An Approach to Architectural Analysis of Product Lines," *Proceedings of the 22nd International Conference on Software Engineering (ICSE '00)*, pp. 548-557, 2000.

[9] JDOM Organization. http://www.jdom.org/, last accessed: 1 October 2003, last updated: unknown.

[10] M. Svahnberg and J. Bosch. "Characterizing Evolution in Product Line Architecures," In N. Debhath and R. Lee, editors, *Proceedings of the 3rd Annual IASTED International Conference on Software Engineering and Applications*, pp. 92-97, Anaheim, CA, October 1999.

[11] World Wide Web Consortium. "XForms 1.0 Reference," http://www.w3.org/TR/xforms/index.html, last accessed: 2 June 2003, last updated: unknown.

[12] X-port. http://www.formsplayer.com/, last accessed: 1 October 2003, last updated: unknown.