An approach to specifying software frameworks

Leesa Murray David Carrington Paul Strooper

School of Information Technology and Electrical Engineering
The University of Queensland Qld 4072
Australia

{leesam, davec, pstroop}@itee.uq.edu.au

Abstract

A framework is a reusable design that requires software components to function. To instantiate a framework, a software engineer must provide the software components required by the framework. To do this effectively, the framework-component interfaces must be specified so the software engineer knows what assumptions the framework makes about the components, and so the components can be verified against these assumptions. This paper presents an approach to specifying software frameworks. The approach involves the specification of the framework's syntax, semantics, and the interfaces between the framework and its components. The approach is demonstrated with a simple case study.

Keywords: Software frameworks, specification.

1 Introduction

This paper presents a language-independent approach to specifying software frameworks. Throughout the literature on frameworks, it is stated that frameworks need to be specified and components need to have "usage descriptions" (Meyer 2003). UML has been extended to enable the modelling of variation points (what we call software components) in frameworks (Fontoura et al. 2000). However, framework specification involves more than specifying the components and their interfaces with the framework. The framework's syntax and semantic behaviour must also be specified. The approach presented in this paper addresses all of these issues.

In the plethora of literature on frameworks, patterns, and components, there are varying definitions of framework and component. To alleviate misinterpretations, we provide definitions for both before describing our approach.

A Framework is:

> *a reusable design that requires software components to function.*

The user of a framework (a software engineer) must provide the code (software components) that the framework calls. This is in contrast to the use of a toolkit or software library, where the software engineer writes the code that calls the code to be reused from the toolkit or library (Gamma et al. 1994). To instantiate a framework, the software engineer must provide a set of components that implement concrete realisations of the design (Schmidt and Buschmann 2003).

A Component is:

> *a software unit (for example, a class, module, or function) or data unit that has a defined interface for which it (the component) provides an instantiation.*

This is an adaptation of Pasetti's (2002) definition of a component. We explicitly include data, or non-software components, in the definition, as these can be components of a framework.

A framework is instantiated when a software engineer provides the concrete components that the framework expects. To do this, the interface of each component and the framework must be specified, so the engineer knows what the framework expects of the component. This is true for software components and data components. When development of the components is complete, it should be possible to show that the components satisfy their defined interfaces.

The specification of toolkits and software libraries, and their use, appears to be well-understood. While frameworks provide another avenue for domain-specific reuse, there is evidence that they are hard to understand and use (Fayad and Schmidt 1997, Baumer et al. 1997). Therefore, engineers tend to re-invent solutions rather than use an existing framework.

To understand and use a framework, the framework must be specified. The engineer must understand what the framework does (that is, its semantics), what components must be provided to instantiate the framework (that is, the interfaces of the framework and each of its components), and how to invoke the framework (that is, its syntax).

Our approach includes both *specifying a framework* and *instantiating a framework*.

Specification of frameworks includes three tasks:

a. Specify the syntax of the framework

b. Specify the semantics of the framework

c. Specify the framework-component interfaces

Copyright © 2004, Australian Computer Society, Inc. This paper appeared at the 27th *Australasian Computer Science Conference*, The University of Otago, Dunedin, New Zealand. Conferences in Research and Practice in Information Technology, Vol. 26. V. Estivill-Castro, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

The syntax of the framework specifies how the engineer uses the framework, that is, how the framework becomes a system or part of a system. For example, if a framework provides a main() or similar function, the engineer must know that this is how the framework code is invoked. The semantics of the framework specify what functionality it provides. The framework-component interfaces define the syntax of the components (both software and non-software) that must be provided to instantiate the framework. If the framework expects the components to provide certain functionality, these semantic properties must be stated as well.

The instantiation of a framework consists of two tasks:

a. Specify the system properties

b. Provide the concrete components

Specification of system properties means defining the system that executes when the framework is instantiated. Just providing a framework with concrete components may form an executable system. At the other extreme, a framework instantiated with concrete components may just form a part of a larger system. In framework specification, we defined the framework-component interfaces, and now in framework instantiation, we provide the concrete components. This allows the engineer to verify that the concrete components have the interfaces and functionality that the framework expects. Showing that the framework with the instantiated components satisfies the system properties is then a necessary verification task, which is not discussed further in this paper.

Our approach to framework specification and use is described in Section 2 and illustrated in Section 3 by a simple case study. The problem of specifying large frameworks is discussed in Section 4. Section 5 presents related work, focussing on other approaches to specifying software frameworks. Section 6 discusses future work and concludes. Throughout this paper we use arial font when writing or referring to source code.

2 An approach to specifying software frameworks

Our approach includes specifying a framework and instantiating a framework. The result of specifying a framework is a description of what the framework does. It can be thought of as an abstract view of the framework and is not expected to change unless the framework undergoes a change of functionality. That is, a framework's specification is generic. This is in contrast to framework instantiation, where the generic specification is parameterised by component realisations, and results in a description of the actual use of the framework. As this is specific for each application, the resulting description is unique for each specific use.

2.1 Specification of the framework

Specifying a framework consists of three tasks:

a. Specify the framework's syntax

b. Specify the framework's semantics

c. Specify the framework-component interfaces

The framework's syntax specifies how a software developer actually uses or invokes the framework. A framework when instantiated with concrete components can sometimes form an executable system on its own. In this case, the framework must provide some method of executing the system, such as a main() operation if the implementation language is C or C++. The specification of the framework's syntax would state that the framework supplies a main() operation. In the case where the instantiated framework does not result in a standalone system, but rather must be incorporated into a larger system, the framework must provide code that is called by this larger system. The specification of the framework's syntax would state what this code is, that is, the name/s of any operations that must be called, along with the specification of any parameters that must be provided. Specifying the syntax of a framework is crucial to its adoption, as a software engineer cannot use a framework if it is not clear how to invoke it.

To specify a framework's semantics means to state explicitly what the framework does and what functionality it provides. It is from this specification that a software engineer can decide if the framework is suitable for their application.

The third task is to specify the interfaces between the framework and its components. This involves stating explicitly each component that must be provided for the framework to function. For each software component, its syntax and any required semantic properties must be specified. That is, the static interface must be specified so the software developer knows how the framework invokes the component. If the framework expects the component to have any specific behaviour, then this functionality must also be specified. For each non-software component, such as a data file, its syntax and any semantic properties that the framework expects the component to satisfy, must be specified as well. The syntax states the format or layout of a data file, and its interface to the framework, for example, a file might have to have a specific name or file extension. Semantic properties of a non-software component state what the component provides, for example, the meaning of the contents of a data file. If there are any relationships between components, these must also be specified as part of this task. The specification of the framework-component interfaces is abstract. By abstract we mean that the specification is from the viewpoint of the framework, and only includes the properties of the components that the framework relies on. Later, when the components are instantiated, the software engineer can verify that the concrete components satisfy the abstract specifications that result from this task.

2.2 Instantiation of the framework

Framework instantiation states the application-specific use of a framework, and consists of two tasks:

a. Specify the system properties

b. Provide the concrete components

The system properties define the system that executes when the framework is instantiated with all of the components it requires. One of the complexities in framework use is ensuring that the framework is applied to areas for which it is suitable. Explicitly stating the system properties allows the engineer to check that the framework application is suitable. Stating system properties also allows for verification tasks to be undertaken to ensure that the framework with its instantiated components satisfies these properties.

The second task is to provide concrete components that are instantiations of the components specified as part of framework specification. For software components, the instantiation is the code for the component. Once the code is supplied, it is possible to verify that the instantiated software component meets its specification. For non-software components, such as a data file, the concrete component is the actual file with the data in it. Similarly, we can check the data file and its contents against its specification. Therefore, there should be a mapping from each concrete component to its specification.

3 A simple case study

Our case study is a graph traversal framework that originated from ClassBench (Hoffman and Strooper 1997), a testing framework. The graph traversal framework allows an engineer to control what happens during the traversal of a graph. The framework traverses the graph in such a way that arc coverage is achieved.

The framework requires two components: a graph and a Driver class. The graph is a non-software component – it is just data. The graph has labelled arcs and nodes, and one distinguished start node. The Driver class is a software component that must provide an implementation for three operations: reset(), arc(), and node(). As an example, Figure 1 shows a graph. The shading indicates that node N1 is the start node of the graph.

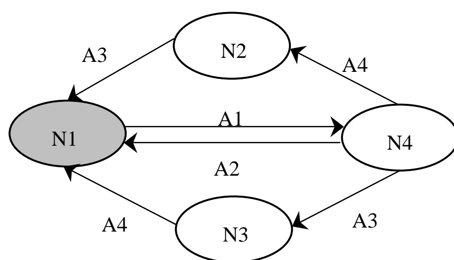

Figure 1. Example graph

Figures 2 and 3 show a Driver class (written in C++). For simplicity, in the Driver class, we have chosen to just print the labels of the arcs and nodes each time they are traversed or visited. The string parameters for the arc() and node() operations correspond to the arc labels and node labels in the graph, respectively. Each time an arc is traversed in the graph, the arc() operation is called with the arc's label. Similarly, each time a node in the graph is visited, the node() operation is called with the node's label.

```
class Driver:
{
public:
        Driver();
        void reset();
        void arc(string);
        void node(string);
};
```

Figure 2. Example Driver header – driver.h

```
Driver::Driver()
{
}

Driver::reset()
{
        printf("\n New Path: \n");
}

Driver::arc(string arcLabel)
{
        printf("arc - %s \n", arcLabel);
}

Driver::node(string nodeLabel)
{
        printf("node – %s \n", nodeLabel);
}
```

Figure 3. Example Driver implementation – driver.cpp

The following sections specify the graph traversal framework and instantiate the framework by providing concrete representations of its components.

3.1 Specification of the graph traversal framework

As stated above, there are three parts to specifying a framework: the syntax, the semantics, and the framework-component interfaces. Each part of the graph traversal framework is specified in turn.

3.1.1 Syntax of the framework

This part of the framework's specification states how the graph traversal framework is invoked. The framework has a main(…) operation. When the system's execution is initiated, the framework's main(…) operation executes. This execution is responsible for: loading the graph (from a file supplied as a command-line parameter), creating an instance of the Driver class, and calling the graph traversal algorithm.

F1 The framework has a main(…) operation that:

a. loads the graph from a file that is supplied as a command-line parameter;

b. creates an instance of the Driver class; and

c. implements the graph traversal algorithm.

3.1.2 Semantics of the framework

This part of the framework's specification states what the graph traversal framework does. The framework provides a graph traversal algorithm that guarantees arc coverage. The framework traverses the graph executing the implementation specified by the user in the Driver class's arc() operation, each time an arc in the graph is traversed, and executing the implementation specified by the user in the Driver class's node() operation, each time a node in the graph is visited.

F2 The framework implements a graph traversal algorithm that generates paths that achieve arc coverage of the graph.

F3 Each graph path begins at the start node.

F4 For each path the framework calls the appropriate Driver operations:

a. the reset() operation is called at the start of the path;

b. the arc(string) operation is called as each graph arc is traversed, with the arc's label as the parameter; and

c. the node(string) operation is called as each graph node is visited, with the node's label as the parameter.

3.1.3 Framework-component interfaces

This part of the framework's specification defines the interfaces between the graph traversal framework and its components, a graph and a Driver class, as well as any requirements of the interfaces between the graph and the Driver class. It states the expectations that the framework has of these components.

The user of the framework must provide a graph with labelled nodes and arcs. Graph nodes correspond to states and the arcs to the transitions between states; there is one distinguished start node. Every path traversed through the graph begins at the start node.

F5 A graph has exactly one start node.

F6 Each graph node has a unique label.

This means that if two nodes in a graph have the same label they are considered to be the same node.

F7 Each graph arc has a label.

F8 Each graph arc has a source node and a destination node.

F9 There is at most one arc (in each direction) between any two nodes in a graph.

That is, in one direction, there can only be one arc between two nodes. As Figure 1 shows, there are two arcs, A1 and A2 between nodes N1 and N4, but these arcs are in opposite directions, which is permitted.

F10 Each graph arc is traversable from the start node.

That is, each arc in a graph must be reachable from the start node.

F11 Each graph node is reachable from the start node.

F12 A graph is defined in a text file where:

a. the first line of the file is the label of the start node; and

b. each subsequent line in the file specifies an arc and its source and destination nodes, in the format: <arc label> <source node label> <destination node label> (without the angle brackets).

Given the format of a graph's file specified in F12, properties F5, F6, and F8 are implied. While we insist that nodes have unique labels, we do not place this restriction on arcs. This decision may seem arbitrary but stems from the original application of the framework, in which the node labels are used for checking states and arc labels for causing state transitions (Hoffman and Strooper 1997).

The user must provide a Driver class, written in C++ with a header file (.h file) and an implementation file (.cpp file). The operations in this class are called by the framework, through the graph traversal algorithm, as a graph is traversed.

F13 The Driver class is implemented in C++ and must have:

a. a header file called driver.h; and

b. an implementation file called driver.cpp.

F14 The Driver must provide the operations:

a. reset();

b. arc(string arcLabel); and

c. node(string nodeLabel).

The string parameters of the arc() and node() operations correspond to the labels of the arcs and nodes in the graph.

3.2 Instantiation of the graph traversal framework

There are two parts to instantiating the graph traversal framework: the specification of the system properties, and the provision of concrete instances of a graph and a Driver class.

In the case of the graph traversal framework, once the framework is instantiated with concrete instances of a graph and a Driver class, it becomes a system. This is because the framework provides its own main() function that executes the system. To illustrate the adaptability of frameworks we specify two different systems that both use the graph traversal framework.

3.2.1 System 1 – a graph traversal system

What we describe here deviates slightly from the traditional use of a framework, however, it does illustrate the versatility that frameworks possess.

Suppose as a software engineer you are asked to implement the system described by the following property.

S1 The system reads in a graph and prints a sequence of paths in the graph that achieves arc coverage.

You are aware of the graph traversal framework specified in Section 3.1, and know that it provides the functionality you require. However, you also know that this framework requires two components, a graph and a Driver class to function. Your graph traversal system is to read in any graph so you do not wish to instantiate the graph component, as the resulting system will then just always read in this specific graph and print out a sequence of its paths. So, can you achieve your task and still use the graph traversal framework? The answer is yes.

If we instantiate the Driver class with the concrete instance of a Driver shown in Figures 2 and 3, the result is a graph traversal system. This is because all of the software components that the framework requires have been instantiated. The concrete Driver component must satisfy F13 and F14. Upon inspection we can see that the header file, driver.h, and the implementation file, driver.cpp, exist, and are written in C++. Therefore, F13 is satisfied. F14 is satisfied as the Driver class does provide the reset(), arc(string arcLabel), and node(string nodeLabel) operations.

The result of this partial instantiation is a system, and this system's specification is a specialisation of the graph traversal framework's specification. The system's specification consists of properties F1 (parts a and c), F2 (augmented), and F3. All properties that mention the Driver are removed because the Driver is now part of the system, and that is why F1 (part b) and F4 have not been included. For the system, F2 is augmented to state that the system implements a graph traversal algorithm that generates and prints paths that achieve arc coverage.

The resulting system takes a graph (in a text file) as a command-line parameter and prints the label of each arc as it is traversed and the label of each node as it is visited, providing the graph complies with the specification described by properties F5 to F12.

Figure 1 is a graphical representation of a concrete instance of a graph, and Figure 4 is a text file representation of the graph in Figure 1. This is an example of a graph that we could use with the graph traversal system as long as we can show that it satisfies properties F5 to F12. Upon inspection of the graph's file and its graphical representation shown in Figure 1, we can see that all properties are satisfied. As an example of the inspection, we verify F11.

F11 Each node is reachable from the start node, N1. There are four nodes in the graph. N1 is the start node. N4 is reachable from N1, via A1. N2 is reachable from N1 via A1 and A4. N3 is reachable from N1 via A1 and A3. Therefore, F11 is satisfied.

```
N1
A1 N1 N4
A2 N4 N1
A3 N2 N1
A3 N4 N3
A4 N4 N2
A4 N3 N1
```

Figure 4. A graph's text file

3.2.2 System 2 – a class testing system

The second system that uses the graph traversal framework is a class testing system that has the following property.

S2 The system tests a C++ class by traversing the paths in a graph and executing state changes implemented in Driver::arc() and checks implemented in Driver::node().

This system is derived from the original use of the graph traversal framework in ClassBench (Hoffman and Strooper 1997). In this instantiation of the graph traversal framework, we are testing the implementation of an integer set class. To do this we must provide a graph representation of the class. Because we are testing a class, we choose interesting state transitions for the arcs of the graph that will exercise the class's operations thoroughly, and specific states for the nodes. In Driver's arc() operation, we call the set class's operations to perform the state transitions that will occur when the framework traverses an arc in the graph. In the Driver's node() operation we provide code that checks that the set's state is consistent with the expected behaviour for the current node in the graph.

Figure 5 shows a graphical representation of a graph that we will use to drive the testing. The shaded node EMPTY, is the start node. The sets shown for each node indicate the contents of each set, and MAX is the maximum number of integers in a full set. Note that this graph is isomorphic to the one shown in Figure 1.

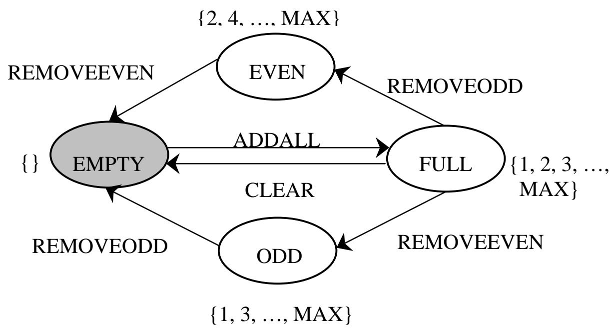

Figure 5. Integer set graph

Figure 6 is a text file representation of the graph component shown in Figure 5.

```
EMPTY
ADDALL EMPTY FULL
CLEAR FULL EMPTY
REMOVEODD FULL EVEN
REMOVEODD ODD EMPTY
REMOVEEVEN EVEN EMPTY
REMOVEEVEN FULL ODD
```

Figure 6. Text file for integer set graph

Upon inspection of the graph's file shown in Figure 6, we can see that all properties specified for the graph component, that is F5 to F12, are satisfied.

The concrete Driver component's header file is shown in Figure 7. As in the graph above, MAX is the maximum number of integers in a full set.

```
const int MAX = 100;

class Driver:
{
public:
        Driver();
        void reset();
        void arc(string);
        void node(string);
private:
        IntSet cut;
}
```

Figure 7. Driver header for the testing system – driver.h

A partial implementation of the concrete Driver component is shown in Figure 8. IntSet is the class being tested, and therefore Driver creates an instance of this class in its constructor. IntSet provides the operations add(), remove(), removeAll(), find(), and size(). Driver's reset() operation deletes all elements from the set so each new path through the graph begins with an empty set. The arc() operation adds and removes elements to and from the set, depending on which arc is traversed in the graph. We have just shown the implementation for arc ADDALL to give a flavour for how the operations of the class under test are called to perform the state transitions represented by each arc label. The node() operation checks that the set's state is consistent with the state expected at each node in the graph. We have shown part of the implementation for node EMPTY, which checks that the size of the set is zero when this node is visited in the graph. For this node we would also check that find() returns false, and for the other nodes similar checks would be implemented.

```
Driver::Driver()
{
    IntSet cut = new IntSet(MAX);
}

Driver::reset()
{
    cut.removeAll();
    printf("\n New path - \n");
}

Driver::arc(String arcLabel)
{
    if (strcmp(arcLabel, "ADDALL") == 0)
        for (int i = 1; i <= MAX; i++)
            cut.add(i);
    else if (strcmp (arcLabel, "CLEAR") == 0)
        …
    else if (strcmp(arcLabel, "REMOVEODD") == 0)
        …
    else if (strcmp(arcLable, "REMOVEEVEN") == 0)
        …
}

Driver::node(string nodeLabel)
{
    if (strcmp(nodeLabel, "EMPTY") == 0)
        if (cut.size() == 0)
            printf("node EMPTY – success \n");
        else
            printf("node EMPTY – FAIL \n");
    else if (strcmp(nodeLabel, "FULL") == 0)
        …
    else if (strcmp(nodeLabel, "EVEN") == 0)
        …
    else if (strcmp(nodeLabel, "ODD") == 0)
        …
}
```

Figure 8. Driver implementation for the testing system – driver.cpp

The concrete Driver component for the testing system's instantiation of the graph traversal framework, shown in Figures 7 and 8, must satisfy F13 and F14. Upon inspection of the Driver code we can see this is the case.

3.2.3 Summary

We have shown that our concrete graph and driver components for both instantiations of the graph traversal framework satisfy their specifications. In this case the verification task was done by inspection and made easy by the fact that the components' properties were precisely

specified. The verification techniques chosen for this task will depend on the complexity of the components, but this task is made more straightforward if an effort is made to specify the component's syntax and any semantic properties as precisely as possible.

The instantiation of a subset of a framework's components to produce a system is possible if all of the software components are instantiated. This is illustrated by the graph traversal system above. If only some of the framework's software components are instantiated, the result is still a framework that may then provide the functionality required to solve another problem. Software engineers must not discount a framework too quickly, but need to apply some lateral thinking if they find a framework that almost provides what they require. The verification of partially instantiated frameworks is a task for future work.

4 Large frameworks

A framework, no matter what size, has to be specified if it is to be used. As already stated, engineers tend to re-invent solutions rather than using a framework, and part of this is due to not being able to determine if the framework is applicable to the problem, and not being able to determine how to use the framework. Therefore, as with all specification, the granularity of the requirements needs to be correct for the size of the framework.

A large framework needs to be specified with requirements of a much coarser grain than those presented for the graph traversal framework. A framework can consist of many parts, including other frameworks and subsystems that are not frameworks. To determine if a framework is suitable for the problem at hand, regardless of its size, the software engineer must be aware of the framework's syntax, semantics and interfaces.

This argument is supported by the first author's experience in a large project involving the development and use of multiple frameworks. The development team met each week and worked closely together, so everyone was aware of what the frameworks were meant to do. With one framework in particular, no specification was provided. That is, the engineers trying to use the framework knew what it was supposed to do, but had no specifics of what components they had to provide or of how they called the framework. Some looked at the source code to work it out, and then used a trial and error approach to get their code running. Others argued that it was too much trouble and they would be better off just writing their code from scratch and not using the framework.

To date our approach to framework specification has not been applied to a large framework, but we plan to trial it on an industrial case study.

5 Related work

There is an abundance of literature on frameworks and components. Here we discuss literature related to the specification of software frameworks. The most closely related work is that of Fontoura et al. (2000) and Bouassida et al. (2003). Fontoura et al. provide an extension to UML called UML-F, for designing software frameworks. They introduce notation that allows variation points (what we call software components) to be explicitly identified, so it is clear that the engineer must provide implementations of these variation points to instantiate the framework. Our approach differs from theirs in two ways. The first is that it is notation-independent. While we agree with Fontoura et al. that the specification of software components and their interfaces with the framework is important, this is just part of framework specification. Other aspects of framework specification such as the framework's syntax and semantics are also important, and these are included in our approach.

Bouassida et al. (2003) also present an extension to UML for designing frameworks, called F-UML. They introduce tags and graphical annotations to distinguish between the core of the framework (the part that does not change), and the whitebox and blackbox hotspots (what we term components). Like UML-F, this helps the engineer identify what they must provide implementations for. The authors define well-formedness rules, which are syntactic rules that are used to guide the construction of a correct design. This work differs from our approach in that it is notation-dependent, and it is not clear how a framework should be specified. What F-UML does provide, is a set of constructs that can be used to specify a framework using an approach like ours, as with F-UML both syntax and semantics can be specified.

Catalysis (D'Souza and Wills 1999) is a UML-based approach to component-based development. It focuses on modelling and specifying components using UML and the Object Constraint Language (to specify semantics), to make reuse of components possible. Frameworks are discussed as a method of design reuse, however no detail is given of what a framework specification should contain, or how to identify the components that the framework needs to function. While instantiation of components and the verification of components satisfying their abstract specifications is discussed, instantiation of frameworks is not.

Fayad et al. (1999) discuss the specification and application of frameworks. They state that the specification provided by the framework designer binds the application developer using the framework, to ensure that the components provided meet their specifications. We agree with this, and make the assumption in our approach that if the developer provides components that can be verified to meet their specifications, then the framework is guaranteed to behave as specified. Fayad et al. limit framework specification to stating that software components should have specifications including preconditions, postconditions, and invariants (where applicable). Our approach is precise about how a framework should be specified, and while we do not insist on the more formal approach of pre- and postconditions for software components, we are explicit about the syntax and any semantic behaviour of software

and non-software components being specified. It is difficult to reason about how to specify the pre- and postconditions of a non-software component, which we allow. We are also explicit about the specification of a framework's syntax and semantics, and the framework-component interfaces. This does not appear to be included in the specification of frameworks defined by Fayad et al.

Pasetti (2002) proposes a rigorous approach to framework specification, that is language-independent. The requirements (termed *functionality* by Pasetti) are at a much higher level and more coarse-grained than we propose, and there are no specific guidelines about what should be specified. The approach does include *mappings* between types of requirements and the entities that implement (or refine) them, and *constraints* on these mappings. This is more formal than what we currently propose, and we could extend our approach with this idea, for semantic properties, to aid in the verification effort. Pasetti states that the framework's functionality must be described, and that this description is intended for the application engineers who are using the framework as a starting point for their development. We agree with Pasetti, and believe that a framework specified using our approach will provide a precise description from which an engineer will be able to decide if the framework is suitable for their application, and enable them to use the framework if it is.

6 Conclusions and future work

This paper describes a language-independent approach to software framework specification. As discussed in Section 4, others have presented notation-dependent approaches and raised issues about framework specification, but these all lack a precise description of what should be modelled or included in a framework's specification. Our approach does this, while not relying on a particular language. One area of future work is to investigate the use of a formal language to specify the syntax and semantics of frameworks and their components.

The next step is to extend the approach to verification by showing how we can verify the system properties using the framework's specification and the specification of its components. This is of particular importance when frameworks are used to build safety-related or mission-critical systems, and builds on our work in Murray et al. (2002) in which we verified a system in terms of its components. As our case study illustrates, the same framework can be instantiated many times, and the resulting systems can be very different. This adds to the verification challenge.

Acknowledgements

The authors wish to thank Alena Griffiths for her comments on previous drafts of this paper. This work is supported by the ARC Strategic Partnerships with Industry – Research and Training (SPIRT) grant C10027025, an Australian Postgraduate Award (Industry), and Foxboro Australia.

7 References

Baumer, D., Gryczan, G., Knoll, R., Lilienthal, C., Riehle, D. and Zullighoven, H. (1997): Framework development for large systems. *Communications of the ACM*. 40(10):52-59.

Bouassida, N., Ben-Abdallah, H., Gargouri, F. and Ben Hamadou, A. (2003): Formalizing the Framework Design Language F-UML. *Proceedings of the 1st International Conference on Software Engineering and Formal Methods (SEFM 2003)*. IEEE Computer Society, 164-172.

D'Souza, D. and Wills, A. (1999): *Objects, components, and frameworks with UML: The Catalysis Approach*. Addison-Wesley.

Fayad, M., Schmidt, D. and Johnson, R. (1999): *Building application frameworks – object-oriented foundations of framework design*. John Wiley and Sons.

Fayad, M. and Schmidt, D (1997): Object-oriented application frameworks. *Communications of the ACM*. 40(10):32-38.

Fontoura, M., Pree, W. and Rumpe, B. (2000): UML-F: A modeling language for object-oriented frameworks. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'2000*). Lecture Notes in Computer Science, 1850. Springer-Verlag. 63-82.

Gamma, E., Helm, R., Johnson, R. and Vlissedes, J. (1994): *Design Patterns*. Addison-Wesley.

Hoffman, D. and Strooper, P. (1997): A methodology and framework for automated class testing. *Software Practice and Experience*. 27(5):573-597.

Meyer, B. (2003): The grand challenge of trusted components. *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*. IEEE Computer Society. 660-667.

Murray, L., Griffiths, A. and Strooper, P. (2002): OptoNet – a case study in using rigorous analysis techniques to justify a revised product assurance strategy. *Proceedings 8th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'02)*. IEEE Computer Society. 232-237.

Pasetti, A. (2002): *Software frameworks and embedded control systems*. Lecture Notes in Computer Science, 2231. Springer-Verlag.

Schmidt, D. and Buschmann, F. (2003): Patterns, frameworks, and middleware: their synergistic relationships. *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*. IEEE Computer Society. 694-704.