Domain analysis: Integration of verification tools

Jeroen Kleijn

December 27, 2014

1 Rationale

Advances in chip technology have led to increasingly more complex chip architectures. This raise in complexity has forced researchers in the field of modern chip architecture to search for techniques which allow them to better cope with this complexity. One of these techniques is xMAS[1, 2], a visual modeling language that allows the description of system-on-chip designs using high-level constructs.

The WickedXmas[3] tool was created to support the design of these network models. Initially this tool primarily allowed visual diagramming of xMAS networks. Over time, the tool has been extended with verification tools that allow static analysis of the designs.

Like many other software projects, the WickedXmas tool has suffered from a decay in software quality due to continuous modifications and the addition of new features. Integration of the design part of the tool and the verification tools is suboptimal. Using the verification tools is a cumbersome process for the end user requiring many steps to perform. Furthermore, from a software architecture point of view, the tool is hard to maintain and extend. Many dependencies on external libraries and utilities severely hamper portability and deployability of the software.

In order to solve above mentioned issues, a complete rewrite of the WickedXmas tool is warranted. Work has already been started on refactorization of the available verification tools. One of the key design issues involves the extensibility of the software. It should be easy to code and integrate new verification tools. In addition, the design tool and the various verification tools should integrate well.

To achieve this level of integration, the source code of the new verification tools has been examined as part of this domain analysis. After a brief overview of the current WickedXmas tool, this document will describe the core classes of the new verification tools. In order to illustrate how these classes are used, one of the verification tools, the Combinatoric Cycle Checker, is introduced and its implementation is explained.
The next part discusses several issues how to realize the integration of the verification tools and the design tool. A number of alternatives will be presented and the advantages and disadvantages of each are listed. Finally, the last section provides recommendations on which of the listed alternatives are most suitable.

Note: One of the assumptions made during this analysis is that the final design tool will be written in c++. This assumption allows to investigate the ways in which the data structures used by the verification tools can be shared with the design tool. Although this assumption is likely valid, results from other domain analyses might express a preference for an other choice of programming language.

Contents

2 The WickedXmas tool

Before exploring the internals of the source code we shall first provide an overview of the current WickedXmas tool. Specifically, we will briefly highlight the steps required to execute the verification modules.

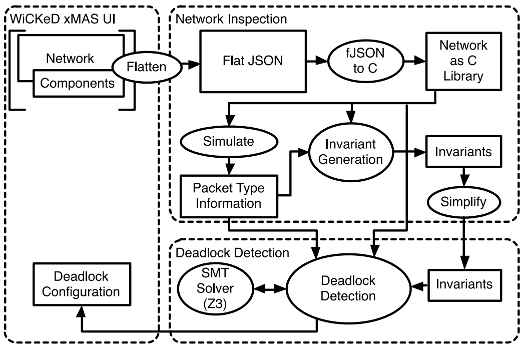

Fig. 3: Tool Architecture

Figure 1: WickedXmas Tool Architecture

Figure 1[3, p. 3] depicts the WickedXmas tool architecture. The design tool (WiCKeD xMAS UI) stores the network in a text file using JSON notation. Before presenting the network to the verification modules (Network Inspection and Deadlock Detection), the network description is flattened. This process transforms the hierarchical network into a flat list consisting solely of xMAS primitives and the channels connecting their ports. The resulting flat json file is then parsed and transformed into C++ source code. The generated source code files are used along with C++ code written for the specific task to create several executables. In Figure 1 the Simulate, Invariant Generation and Deadlock Detection verification modules are shown. The executables must usually be run in a fixed order; results from a module can serve as input for modules further down the pipeline. For instance, the Simulator runs a simulation on the network and yields information on the types of packets each component in the network is able to accept and generate. The Invariant Generation uses this information to deduce invariants that hold for the given network. Finally, the Deadlock Detection module uses both the Packet Type Information and (simplified) Invariants to perform its task and present the results back to the xMAS UI.

Dependencies Several utilities are used while performing these tasks. The conversion from JSON to C++ code is handled by a javascript file and processed by phantomjs. A Haskell program takes care of the Invariant simplification task. To compile the C++ code and link the object files to form the executables a C++ compiler is required. The use of all these different utilities reduces the deployability of the WickedXmas tool. All utilities must first be installed on the target system used to run the tool.

3 New verification modules

In order to improve the software quality the verification modules are currently being refactored. Central to this process is the use of a single set of data structures used by all verification modules. All modules are (or will be) written in C++ (C++ 2011). While designing the new data structures special attention has been paid to ensure extensibility. The data structures should not only be able to provide current needs but also allow the integration of new verification modules. As of this writing, several modules have been ported in varying degrees to use the new data structures. One module that has been completely ported is the Combinatoric Cycle Checker. We will use this module as an example to illustrate the use of the data structures. This module is relatively easy to understand and will therefore serve us well as an example.

3.1 xMAS data structures

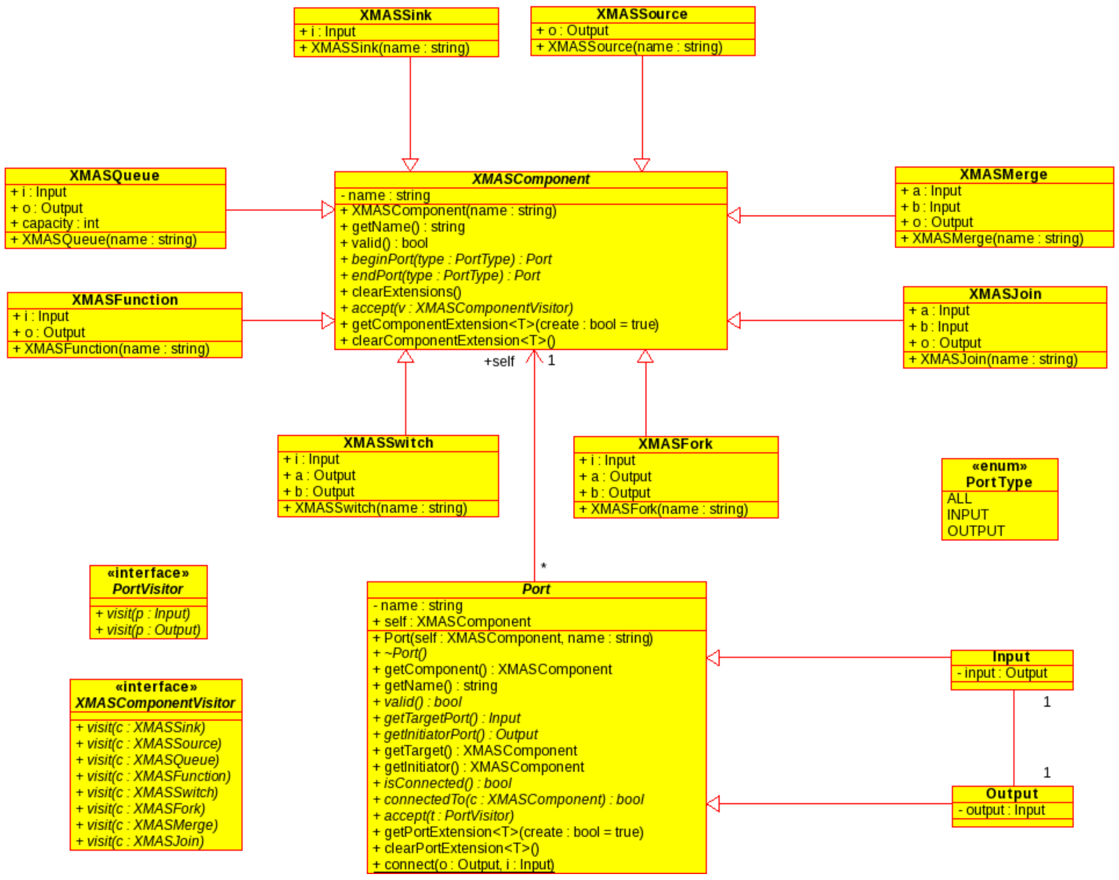

Figure 2: xMAS Class Diagram

In this section we will explore the new data structures. Figure 2 shows a UML-diagram describing the core xMAS data structures. Each component in an xMAS network is represented by an instance of the abstract XMASComponent class. All components can be identified by their unique name. A component has an arbitrary number of ports, the number and type of ports depends on the actual component type. For the currently defined primitives, this number varies between 1 and 3.
A port is represented by the Port class. Like XMASComponent, this class is an abstract base class. Actual instances of Port are either an Input or an Output. Ports are identified by their name just like components. Usually, ports have short identifiers like "a" and "b".

The channels of an xMAS network have no direct class equivalent in the data structures. Rather, a channel is formed by the connect() function. Input and Output both contain a pointer to an object of the opposite type. The connect() function takes an Input and Output and updates these pointers such that they mutually reference each other. The values of these pointers are guarded by an important invariant. At all times, except of course during connect(), an Input should either be disconnected or point to an Output which in its turn should point back to the Input. The same statement should hold when switching the roles of Input and Output.

Looking at the operations defined on Port, we can divide them into several categories.

- The first category of operations deals with the identity and status of the port
  - The Port constructor, destructor, getComponent() and getName() are self-explanatory. They initialize, destroy and query the attributes of a Port.
  - isConnected() returns whether the port is connected to an other port or not.
  - isConnectedTo(XMASComponent) returns whether this port is connected to a port of the specified component
  - valid() checks if the port is connected and, additionally, if the invariant mentioned earlier holds for this port.
- The second category of operations provide information on the channel formed by this port
  - connect() has been explained above, this operation establishes a channel between two ports
  - getTarget() and getTargetPort() return the component and its port that are on the receiving end of the channel. The target port thus is an Input to the component returned by getTarget().
  - getInitiator() and getInitiatorPort() return the component and its port that are on the sending end of the channel. The initiator port thus is an Output from the component returned by getInitiator().

The presence of these methods show the dual role of class Port. It represents not only a port in an xMAS network, but also the channel between two ports, of course provided the port is actually connected. The channel-role is equally well fulfilled by either port.

- The other operations: accept, getPortExtension and clearPortExtension will be explained in detail in the next sections.

Returning our focus to XMASComponent we notice many operations similar to those of Port.

- The constructor and getName() operations set and query the name of the component.
- valid() returns true if all ports of the component are valid and false if at least one port is not valid.
- beginPort() and endPort() return iterators to loop through all available ports. The type arguments can be used to specify if only input, output or all ports should be returned.
- accept and the -Extension functions will be covered in the next sections.

XMASComponent itself is an abstract class. Each of the 8 xMAS primitive types is represented as a separate class inheriting from the XMASComponent base class. Whereas XMASComponent only provides access to ports, the primitive classes actually instantiate them. Some primitive types have additional attributes, e.g. XMASQueue has an attribute to indicate the queue capacity. Maybe surprising is the fact that other primitive types seem to lack the required extra attributes. For example, XMASFunction has no attribute to hold the function to apply to the data packets and XMASSource has no attribute specifying what type of packets it should produce. These seemingly missing attributes are stored using the extension mechanism covered in the next section.

3.2 Extensions

The classes that represent the 8 xMAS primitives and their base class XMASComponent provide few features to support the verification modules. For instance, many modules need to know what packet types a component can accept or produce but neither XMASComponent nor its descendants are able to provide this information. This design is a deliberate choice. The classes only expose core attributes like their name and the ports that are available. However, all components allow extra data to be stored in so called extensions. The same mechanism is available to augment ports with arbitrary extra data.

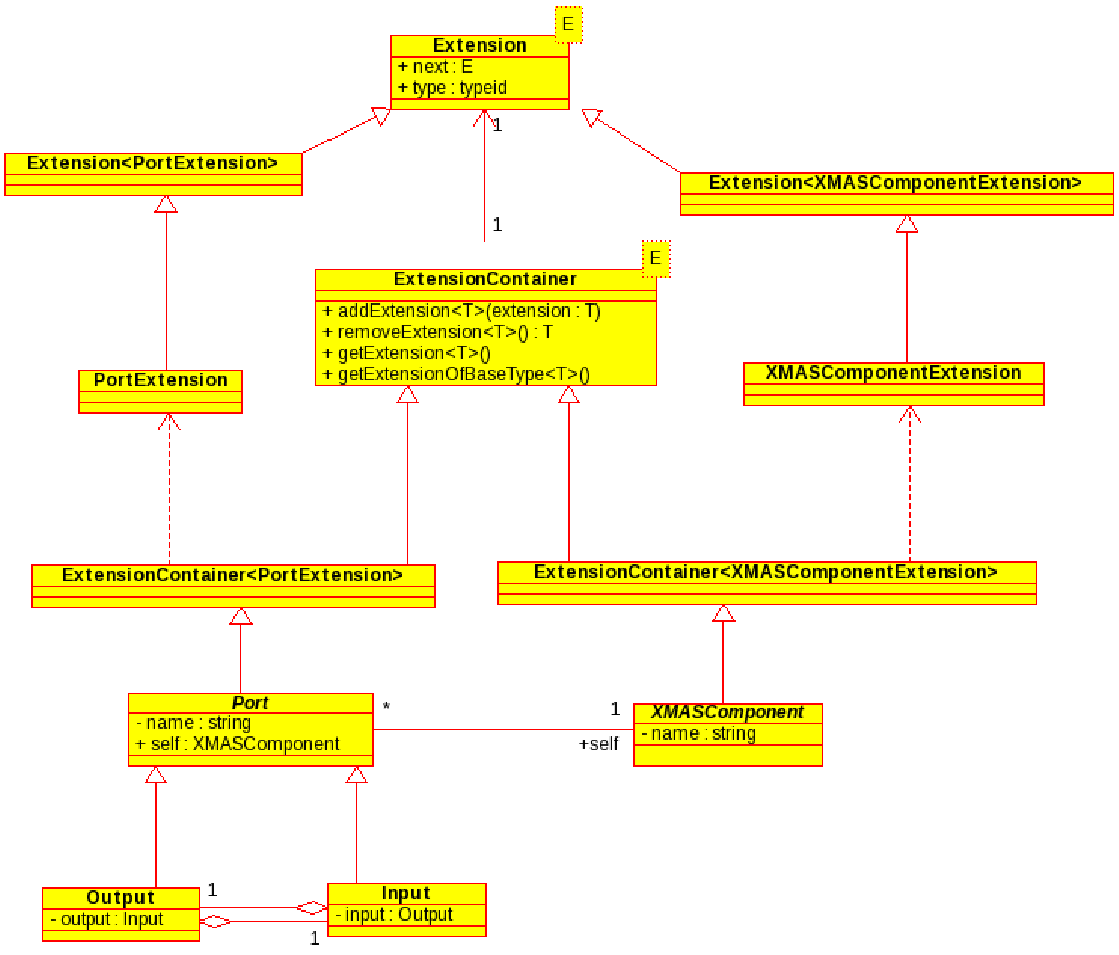

Figure 3: xMAS Class Hierarchy

Figure 3 shows the classes that implement the extension mechanism. Both XMAS-Component and Port are actually part of a relatively complex class hierarchy. At first sight, the extension mechanism appears to be quite complicated. The basic idea, however, is very simple. All ports are able to hold a list of PortExtension objects, likewise, all components are able to hold a list of XMASComponentExtension objects. Any module can define a new Extension class (e.g. CombinatorialCyclePortExtension) and make it a subclass of either PortExtension or XMASComponentExtension, depending on whether the extension stores attributes of a port or of a component. The ExtensionContainer class provides methods to add, remove and fetch the extensions of a given type. For instance, the cycle checker will call getExtension() using CombinatorialCyclePortExtension as the template parameter T.

The linked list of extensions could easily have been implemented using the default STL container classes, like std::list or std::forward_list. The chosen design has a few important advantages. The first one is related to the pointers required to maintain the list. Here, each extension stores the pointer inside the extension itself by inheriting the next attribute from Extension. This approach, called internal storage[5], differs from the approach used by the STL lists. External storage, as used by STL, is the more generic approach as it places no constraints on the value type (see [5] for details). Since the data structures are self-developed and can be designed to incorporate internal storage, the space and run time performance benefits of internal storage outweigh the extra effort.

The second aspect of the extensions design concerns type checking. In the diagram this aspect is reflected through the use of the generic Extension class and its type attribute. The main benefit of this design decision is again performance but additionally it improves type safety.
Each component (or port) holds a heterogeneous list of extensions, each derived from XMASComponentExtension (or PortExtension for ports). By equipping class Extension with a template parameter the next attribute's type can be accurately specified. To illustrate this, let's take a look at the definition of PortExtension:

```
class PortExtension : public Extension<PortExtension>{
}
```

The template parameter E of Extension is bound to PortExtension. The result is that the next attribute that PortExtension inherits from Extension has type PortExtension*. The compiler can now check that the list actually contains extensions derived from PortExtension.

Although the linked list is guaranteed to only contain valid extension types, most verification modules are only interested in specific subclasses of PortExtension or XMAS-ComponentExtension. ExtensionContainer provides methods to efficiently filter the list of extensions. getExtension<T>() returns the first extension whose type matches that of template parameter T. getExtensionOfBaseType<T>() also select extensions that are derived from T. The type attribute defined in Extension aids ExtensionContainer to select the correct Extension.

3.3 Visitor Pattern

The extension mechanism enables verification modules to augment components and ports with arbitrary extra data. Many times it is desirable to extend the behavioural features of the classes as well. The Visitor Pattern provides this behavioural complement to extensions.

When the Visitor Pattern is applied to an object-oriented class design, operations normally defined on the domain classes are pulled out of the class and accomodated in

a new class known as a visitor. The Gang of Four defines the Visitor as:
"Represent an operation to be performed on elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates."[6]
The visitor pattern is essential for the modularity and extensibility of the verification modules. Rather than implementing all verification code inside the domain classes, each module defines one or more Visitor classes. Inside these classes the implementation of the verification code is placed. Adding a new module does not require a single change to XMASComponent, Port or any of the other domain classes.

The Visitor Pattern applied in C++ makes use of two features of the language. The first one is function overloading. Overloaded functions have multiple distinct implementations while all share the same function name. The functions however differ in the number and type of arguments they operate on. When a compiler encounters a call to an overloaded function, it inspects the arguments it should pass and uses this information to resolve the correct function to call.
The second language feature is dynamic dispatch. Through the use of virtual functions the actual function to invoke is determined at run time.
When used together, these features implement double dispatch. Overloaded functions allow differentiation based on the arguments of the function, virtual functions allow differentiation based on the actual type of the calling object.

In the class diagram two Visitor interfaces are defined, one for ports and one for components. Visitors that implement interface PortVisitor must provide two implementation of the overloaded function visit(). One to handle visits to an Input port and one to handle a visit to an Output port. Likewise, implementations of interface XMASComponentVisitor must implement the visit method for each of the 8 xMAS primitive types.

Figure 4 shows the Visitor Pattern in action. CombinatorialCycleDependencies is an implementation of the XMASComponentVisitor interface. In this example, an instance of this class calls the accept method on an XMASComponent instance, in this case an XMASMerge. Through dynamic dispatch the accept method as defined by XMASMerge is called. The implementation of this method is:

```
void accept(XMASComponentVisitor &v) {
  v.visit(this);
}
```

Immediately, control is returned to the Visitor class by calling its visit method. Since the method is defined as part of the XMASMerge class, the *this* pointer passed has type XMASMerge*. The compiler uses this knowledge to infer that it has to call the overloaded visit method which takes a pointer to an XMASMerge component (overloading

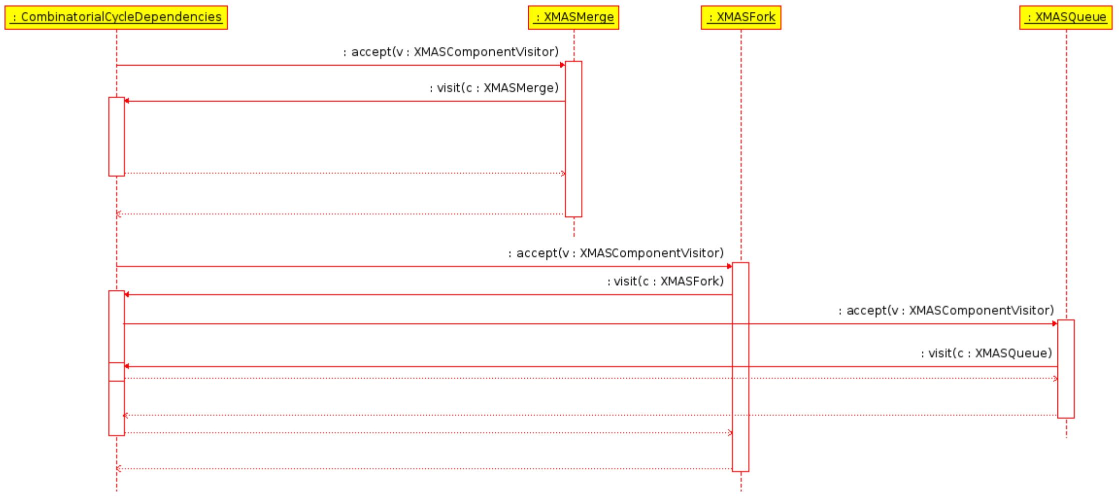
: CombinatorialCycleDependencies
: XMASMerge
: XMASFork
: XMASQueue
: accept(v : XMASComponentVisitor)
: visit(c : XMASMerge)
: accept(v : XMASComponentVisitor)
: visit(c : XMASFork)
: accept(v : XMASComponentVisitor)
: visit(c : XMASQueue)

Figure 4: Visitor Sequence Diagram

and dynamic dispatch are combined here to perform double dispatch). The CombinatorialCycleDependencies class is now able to execute code specifically written to handle an XMAS merge. The diagrams also shows a slightly more complex example where the Visitor recursively calls accept on an XMASQueue instance during the processing of a visit to an XMASFork object.

Open/Closed Principle Both the Extension mechanism and the Visitor Pattern are applications of the Open/Closed Principle. This principle states that: “Software entities like classes, modules and functions should be open for extension but closed for modifications”[8]. Minimizing changes to existing classes that have been tested and used, prevents undesired effects on current functionality. Core classes like XMASComponent and Port are used in many different parts of the software (see also fig. 6), so changes to these classes have impact on the entire application. At the same time, the priniciple dictates that new features can still be added to the software entities.

4 Combinatoric Cycle Checker

The semantics of an xMAS network can be expressed using the concepts of a combinatorial part and a sequential part[4]. The primitives can be divided into two categories. Sources, sinkes and queues are known as sequential primitives. The other five: function, fork, join, switch and merge are combinatorial primitives.
Each clock tick a source can produce a packet, a sink can consume one and a queue can accept and/or release a packet; all as long as the necessary conditions hold. The clock ensures all sequential primitives are updated at the same time.
The combinatorial objects’ task is to route the packets from one sequential primitive to another. Of course, the time required to perform this routing depends on the amount

and types of combinatorial objects along the path from source to destination. But, as far as the simulation of the network is concerned, the entire routing occurs during a single clock cycle.

One important aspect of an xMAS design is the absence of combinatoric cycles. That is, an xMAS network should not contain cycles in the graph consisting of only combinatorial objects. Would this be the case, the irdy and trdy signals of the combinatorial objects will never stabilize. This can be seen by looking at the equations that define the irdy and trdy signals [1]. The source and sink primitives irdy and trdy signals are determined by an oracle (neglecting the *pre* part for simplicity). Queues are defined to always accept data (i.trdy = 1) as long as the queue is not full and release data (o.irdy = 1) as long as the queue is not empty. All of these conditions, the oracles and the queue state, are well-defined and independent of other components in the network.
In contrast, all combinatorial objects' irdy and trdy equations are defined in terms of the irdy and trdy signals produced by the components on the other sides of the channels. To determine an irdy or trdy signal of a combinatorial primitive, the connected components' irdy and trdy signals should be substituted in the equation. Large chains of combinatorial primitives as such yield complex expressions. More so, when a component is part of a combinatorial cycle, the same irdy or trdy signal occurs on both the left-hand and the right-hand side of the equation. Substitution of the right-hand side occurrence of the signal leads to an infinite recursion.

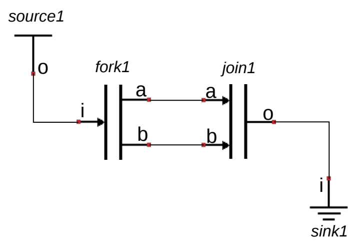

Figure 5: Example of a combinatorial cycle

Example Figure 5 illustrates a combinatorial cycle. Applying the *a.irdy* equation to *fork1* and the *b.trdy* equation to *join1* gives [1, p. 44]:

- fork1.a.irdy := fork1.i.irdy and fork1.b.trdy = source1.o.irdy and join1.b.trdy
- join1.b.trdy := join1.o.trdy and join1.a.irdy = sink1.o.trdy and fork1.a.irdy

Substituting the second equation in the first now gives:

- $\underline{\text{fork1.a.irdy}}$:= source1.o.irdy and sink1.o.trdy and $\underline{\text{fork1.a.irdy}}$

The Combinatoric Cycle Checker The Combinatoric Cycle Checkers job is to detect these combinatorial cycles in an xMAS network. It is based on a standard graph traversal algorithm to detect cycles. For all ports of all components in the model, the algorithm executes a depth-first traversal. At each step, the traversal continues according to the signals on the right-hand side of the equations of the current component. For input ports this equation is the one that defines the trdy signal, for output ports this equation is the one that defines the irdy signal.
Along the traversed path, each port, including the starting port, is marked as 'checking'. If the algorithm later visits this port again, it knows it has detected a cycle. The traversal along a path ends when it reaches a sequential component. During backtracking, all 'checking' marks are reset. If none of the paths starting from a port result in a combinatorial cycle, the port will be marked as 'checked'. The Cycle Checker has completed when all ports have been marked as 'checked'.

Implementation The Combinatoric Cycle Checker is implemented using an extension class, a component visitor and several functions to drive the checker. Class CombinatorialCyclePortExtension is, as the name already suggests, a PortExtension. It holds two boolean attributes: *checking* to indicate whether the algorithm is currently traversing this port and *checked* which is set to true after the port has been successfully checked. CombinatorialCycleDependencies is an implementation of the XMASComponentVisitor interface. The constructor takes a pointer to the Port object that was used to reach the component currently being visited. Attribute p is initialized with this value. A second attribute, *retval*, is an initially empty vector of ports. The various visit member functions will fill this vector with the ports that the algorithm should use to continue the traversal. Visits to the sequential primitives will leave this vector empty. When a combinatorial primitive is visited, all ports required to evaluate the irdy or trdy signal of p are added to the vector.

Inside function checkSignal() the actual algorithm is executed. It is here where the PortExtension is created and the *checking* and *checked* values are set. Also the visitor class is instantiated and used to visit the component. Depending on the contents of attribute *retval*, the algorithm recursively continues its traversal. The Cycle Checker is started by calling CombinatorialCycleDetector and passing the set of all components.

5 Integration of Verification Tools

Ultimately, the verification tools must be integrated with the new WickedXMas design tool. This section provides suggestions how to realize this integration.

5.1 Tool Architecture

In the WickedXMas Tool Architecture the design tool and the verification tools are clearly separated software components. The tools are written in varying languages and they are implemented as separate executables. Now that all tools are being rewritten in a common language (C++), more efficient communication between the verification tools is possible. All tools are integrated into a single executable which is executed as a single process. Due to this, the tools share the same address space and can directly access information produced by other tools.

With regards to the integration with the design tool, two choices are available.

1. The design tool could be integrated completely with the verification module. *(in-process)*

    - Not only the verification tools but also the design tool become part of a single application. Both the design tool and the verification modules operate on the same copy of the xMAS data structures.

2. The design tool and the verification tools are implemented as two separate programs. *(out-of-process)*

    - The design tool launches the verification tools as a new process. Inter-process communication mechanisms are used for communication between the two programs.
    - Both software components have their own representation of an xMAS network. The design tool can not directly access the data structures of the verification tools. However, the design tool can reuse source code, like the classes used by the verification tools (XMASComponent, Port, etc.). (See also section 5.3)

5.2 Module dependencies

The execution time of the modules can vary greatly. When large networks are provided as input to the modules, algorithms with high run time and space complexity may consume many system resources. Therefore, it might be desirable to disable some verification modules if their results are currently not of interest.
The xMAS data structures have been designed with modularity and independency in mind. But, as shown in the WickedXMas Tool Architecture, the verification modules don't operate in a completely isolated fashion.

The order in which the modules are executed is important. Some module can only be executed after the results from other modules are available. In order to provide flexibility in the choice of verification modules to run, careful design of the verification modules is necessary. If possible, the most time consuming algorithms should be easy to disable. That is, other modules should minimize dependency on the results of those modules.

5.3 Using Extensions and Visitors for the design tool

The use of Extensions and Visitors is not limited to just the verification modules. The same mechanisms can be utilized by the design tool. Properties like the position and orientation of an xMAS component on the design canvas can be stored in a new Extension class. The Visitor Pattern can be used for tasks like rendering an image of a component on the canvas. Another use of the pattern is to export xMAS networks to other representations like Verilog.

Composite objects There is however one issue that requires some thought. The verification modules currently have no notion of composite objects. The design tool on the other hand makes heavy use of composite objects to maintain a clear overview of the network.
We briefly explore three approaches to resolve this issue. All approaches assume that a new class, XMASCompositeObject, is introduced to represent a composite object.

In the first approach the XMASComponentVisitor interface is extended with an additional method to visit a composite object. The implementations of XMASComponentVisitor use the methods of XMASCompositeObject to obtain access to the ports and components inside the composite object. This approach is easy to implement. The downside is that all verification modules have to be adjusted to implement the new XMASComponentVisitor interface.

The second approach tries to hide the existence of composite objects altogether from the verification modules. Rather than extending the Visitor interface, the accept method of XMASCompositeObject is implemented in a non-default way. Instead of passing itself as the argument to visit, the composite object calls the accept method on all components inside the composite object, essentially performing the flattening step.
The benefit of this approach is that it requires no changes to existing modules. On the downside, this approach is harder to implement. Care has to be taken that the transparency of composite objects is complete. For example, the objects returned by the getInitiator() and getTarget() methods of Port must not return a XMASCompositeObject object since the modules are likely to use these methods. Possibly, special CompositeInput and CompositeOutput classes are required in order to implement this approach.
Another issue is that transparency is not always desirable. Unlike the verification modules, the design tool sometimes prefers to view a composite object in its entirety while at

other times the design tool wants to access the components inside the composite object. Whether this second approach is feasible remains a question.

A third approach explicitly executes the flattening step. The design tool uses a separate copy of the data structures. Before the network is presented to the verification modules, all composite objects are flattened to create a network without composite objects. Like the second approach, existing verification modules need no modifications. Memory usage will increase though to store the two different views of the network. Also, feedback from the verification modules must be translated back to the original network model with composite objects.

5.4 Feedback from tools

Feedback produced by the tools can be presented back to the design tool in several ways. In this section, the types of feedback and possible ways to present feedback to the design tool are covered.

5.4.1 Types of feedback

Verification modules may want to provide various types of feedback:

- errors found in an xMAS design, for example
  - a combinatoric cycle detected by the cycle checker, the design tool could highlight the components and channels that are part of the cycle.
  - an error in a function specification
- information which might be interesting to the user, for example:
  - invariants
  - packet type information
- progress information
  - Gives the user an indication of the remaining time required by a verification module to verify a network. Not all algorithms are able to provide an accurate progress indication.

5.4.2 Feedback communication

Send feedback to an ostream Currently, the verification modules use the standard output stream (stdout) to send feedback (and debugging information) to the user. By generalizing this method to output feedback to an arbitrary ostream, this mechanism can also be used to stream the feedback to the design tool. In order for the design tool to interpret the feedback, a well defined data format must be agreed upon. This format must be capable of transmitting all types of feedback. Also, a convention to refer to entities like components and ports must be agreed upon.

Define a custom feedback streaming interface Instead of the (character based) ostream, the verification module passes feedback through a streaming interface specifically written for this purpose. Possible operations this interface might expose include:

- info(XMASComponent* component, string message)
- error(Port* port, string message)
- progress(int progress, int total)

Multiple implementations of this interface can be written. For instance, one implementation could simply write the feedback to stdout, using the same output format as above. Another implementation could directly access the data structures passed, e.g. the XMASComponent of the info operation in the example. In this case, the design tool doesn't have to parse the character stream and lookup the component. The design tool and the verification modules must share the same address space for the latter implementation to work correctly though.

Let design tool read extensions defined by modules Unlike the previous methods, the verification modules don't send feedback to the user at all. Rather, the design tool accesses the extensions created by the verification tools to gather information. When using this method, the feedback is pulled by the design tool instead of pushed by the verification modules. As a result, the verification modules need only expose their data through the extensions. The design tool can decide what information it actually needs and fetch it by itself. Detailed and tool-specific information can be communicated without the need to define an interface capable of transmitting all message types.
Drawbacks of this method include the increased coupling between design tool and verification modules. The design tool must know which extensions each verification module uses and write custom code to extract and display the feedback. Another issue is that information like progress is hard to extract from just the extensions. If possible at all, this would require periodic polling of all components/ports in the network to check their status with regards to a verification step.
Again, since the design tool directly accesses the extensions created by the verification modules, the design tool and the verification modules must share the same address space.

Pass feedback to design tool by storing it in an Extension This method also uses the Extension mechanism to give the design tool access to the verification tools' results. The difference is that all feedback is stored in special Extension classes used purely for the purpose of passing feedback. The design tool no longer needs to know about the internals of the verification tool, reducing coupling. Like the previous method, detailed and tool-specific information is available to the design tool, but this time, the verification modules must collect the information and store it in the feedback extension. The same issues with regards to progress information and the “same address space“-restriction apply.

6 Recommendations

Several options are available to integrate the design tool and the verification modules. The recommendations in this section are based on two important use cases:

1. the application will be used to design and verify xMAS networks
2. the design tool will be used to develop and debug verification tools

The use cases lead to different requirements. In order to support the first use case, efficiency of the application is an important aspect. The design tool is an interactive application and should be responsive. When verifying large networks, large amount of memory and processing time can be consumed.
During development of the verification tools, efficiency is of less concern. Mainly small networks are used to debug the tools. More important is the ability to support an efficient 'edit-compile-run-debug' cycle. The source code of the verification modules will frequently change. After a change has been made, the effort required to test the new code should be kept at a minimum.

Tool Architecture Both architecture types discussed in section 5.1, in-process and out-of-process, can be used to implement the two use cases. An in-process architecture leads to more efficient communication between design tool and verification modules. As such, the first use case would suggest to take this approach. As an additional benefit, this approach is easier to implement.
However, an out-of-process might be better suited to support the second use case. After a change has been made to a tool, only the source code related to the verification tools has to be recompiled, reducing the time spent in the compile phase of the 'edit-compile-run-debug' cycle. Furthermore, the design tool doesn't even have to be restarted to test the changes.
Another reason to opt for the out-of-process architecture is the ability to execute the verification tools on an other, maybe more powerful, system.

Both architecture types have advantages and disadvantages. The relative importance of the two use cases can be the decisive factor when choosing between them. Since the intended use and target audience of the tool may change over time, it is recommended to consider the possibility to change the architecture in the future. Therefore, solutions that are only viable in both architecture types are preferred.

Communication Of the four communication methods described above, only the first two can easily be used in an out-of-process architecture. The third and fourth both use extensions and this requires either a shared address space between design and verification tools or an (platform specific) inter-process memory sharing technique. Although the detailed information available in the extensions is an advantage of these methods, simple text messages to describe the feedback are thought to provide enough information. In most cases, the code required to 'push' feedback from the verification modules is already in place which further reduces the advantages of the third method.

The first two methods are both good candidates. The second method has a few advantages, especially when using an in-process architecture. Feedback can be passed more efficient, due to the direct access to the components and ports. This is not very important though as the feedback interface is very low traffic and data structures like hash maps allow efficient lookup of the xMAS objects by id. However, since the second method is capable of producing the same output as the first method and furthermore guarantees a consistent data format output across all modules, the second method is recommended.

Composite objects The best way to deal with composite objects is not clear yet. Ideally, the verification modules will be adapted to handle composite objects. This prevents any communication issues that may arise due to the different network representations of the design tool and verification modules. Modifying the verification modules might prove to be too laborious. In this case, the network must first be flattened. Implicitly using the second approach, or explicitly using the third approach.
The second approach is the technically more interesting approach and has the advantage of a lower memory footprint. However, the simpler third approach is the safer option and therefore recommended.

As a final note, if the design tool will be programmed in a language other than C++, few options remain. Only an out-of-process architecture is possible. Code written to create the verification modules can not be reused, although, of course, the same classes and design patterns could be employed in most other languages. The best option to establish the communication is through a character stream as virtually all platforms and programming languages support this form of communication.

7 Glossary

(verification) module Software component which applies a verification algorithm to an xMAS network

verification tool Same as verification module

design tool Software component to visually model an xMAS network

Visitor class A software class that implements the Visitor interface

References

[1] Chatterjee, Kishinevsky, Ogras, *Quick Formal Modeling of Communication Fabrics to Enable Verification*, 2010

[2] Chatterjee, Kishinevsky, Ogras, *xMAS: Quick Formal Modeling of Communication Fabrics to Enable Verification*, 2012

[3] Joosten, Verbeek, Schmaltz, *WickedXmas: Designing and Verifying on-chip Communication Fabrics*, 2012

[4] *Inference of channel types in micro-architectural models of on-chip communication networks*, 2014

[5] *https://en.wikipedia.org/wiki/Linked_list#Internal_and_external_storage*, november 2014

[6] *https://en.wikipedia.org/wiki/Visitor_pattern*, november 2014

[7] *http://www.oodesign.com/visitor-pattern.html*, november 2014

[8] *http://www.oodesign.com/open-close-principle.html*, november 2014

[9] Stroustrup, *The C++ Programming Language.* Fourth edition, 2013

A xMAS header files

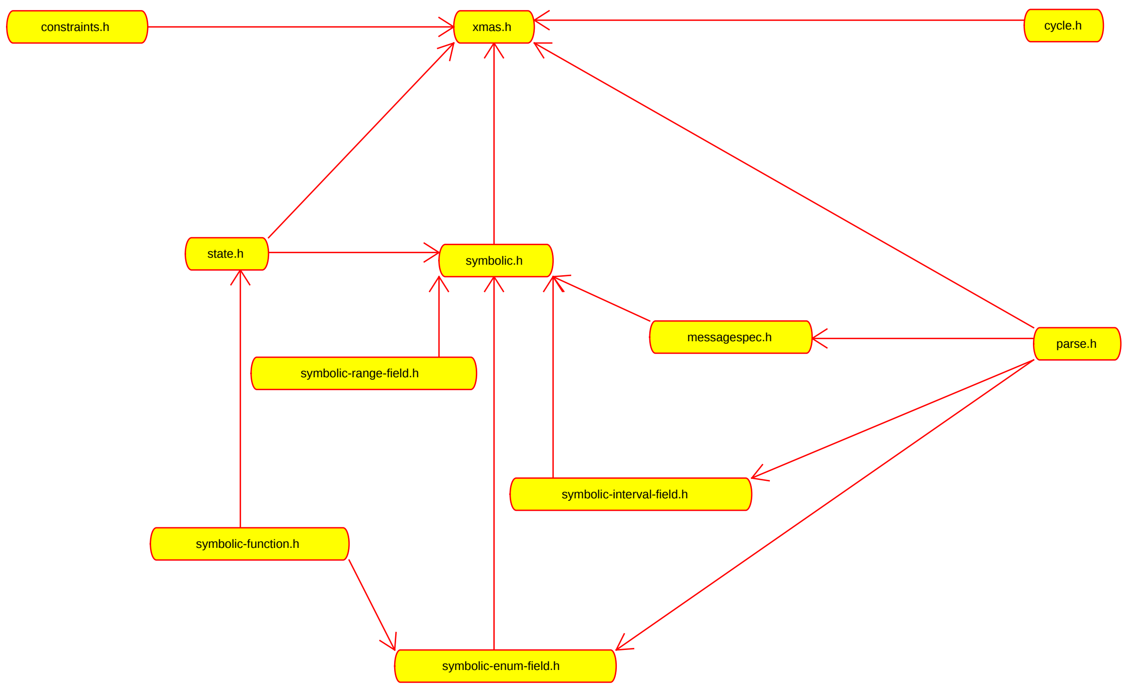

Figure 6: xMAS header files