Exploring Models of Computation through Static Analysis

Ivan Jeukens[1]
Department of Electronic Sytems
University of São Paulo
ijeukens@lme.usp.br

Marius Strum
Department of Electronic Sytems
University of São Paulo
strum@lme.usp.br

Abstract

In this work we present the first version of a tool designed for determining the valid models of computation of an executable specification. This is done by parsing the source code of the specification and checking if the restrictions imposed by a given model of computation are respected. We have successfully analyzed 62 library components out of 103. We also successfully analyzed a benchmark specification.

1 Introduction

The traditional approach for creating an executable specification of a system is based on electing a suitable specification language. The suitability of a language is determined by a match between characteristics of the systems under specification (SuS) and the language's model of computation (MoC). With the necessity of designing heterogeneous systems, novel specifications languages and frameworks have appeared contemplating the modeling with multiple MoC. Among others, Ptolemy II [BHA02] is an example of such framework. It provides the designer with twelve different MoC, and allows for easy addition of new ones.

A typical approach for heterogeneous modeling is based on a general decomposition of the SuS into subsystems and then choosing a MoC for describing each subsystem. This choice is done via elimination of candidate MoCs by general characteristics, such as model of time or the type of interaction with other subsystems and the environment. For example, a subsystem requiring a continuous time model rule out all untimed models, and all models that are unable to interact with a continuous one. This solution is an extension to the traditional single model language approach.

We believe that the choice of MoC should be considered in a design project, since more than one MoC might be suitable for capturing a part of the system. Different MoCs have different characteristics (expressiveness, synthesizability, verifiability, etc), therefore affecting the final quality of the design and its design time. This selection should be done by means of a systematic exploration of different solutions. In fact, there are two interrelated tasks: the decomposition of an specification into different subsystems and the election of a MoC for describing each subsystem. Depending on the decomposition, the use of a particular MoC might be impossible. In the same way, a MoC can restrict the possible decompositions of a system.

In this paper, we describe an approach for selecting models of computation based on statically analysis of the specification's source code. This is done by checking if the restrictions imposed by a certain MoC are met by the specification. We are currently contemplating two models of computation: synchronous data flow (SDF) and synchronous reactive (SR). Section 2 presents a brief description of

1 Funded by the Fundação de Amparo a Pesquisa do Estado de São Paulo (FAPESP), under contract 00/12146-0.

the Ptolemy II framework. Section 3 comments on a related study. Section 4 addresses the major steps of the analysis tool that we have developed. The algorithms for determining the validity of the SDF and SR models for a specification are given in section 5 and 6, respectively. Section 7 addresses the analysis of heterogeneous specifications. Finally, results and conclusions are presented.

2 Background

This work employs the Ptolemy II framework as a tool for creating executable specifications. It is based on the coordinated interaction between components.

At its lower level, it provides the means for describing a clustered graph. The main class is the Entity. Entities can have Ports, that can be interconnected. Each interconnection creates a channel between ports. A hierarchic version of the Entity class is available[2].

On top of this abstract syntax, an abstract semantics is implemented providing an infrastructure for data exchange and execution. For communication, an interface called Receiver contains 12 methods used for data transfer between components, such as:

- *get()*: returns a data from the receiver;
- *put()*: puts a given data into the receiver;
- *hasToken()*: tests for the presence of data in a receiver;
- *hasRoom()*: tests for the availability of storage room in a receiver;
- *isKnown()*: tests if the receiver knows if there are data in it.

The Port class is specialized into the IOPort class. An IOPort specifies the direction of data transfer. A Receiver is associated with every input IOPort.

The Ptolemy II framework divides execution into a number of iterations. An interface called Execution provides 9 methods that are called in a certain order, such as:

- *initialize()*: called once at the beginning of execution;
- *prefire()*: called once in an iteration, before the *fire()* and *postfire()* methods;
- *fire()*: called once in an iteration, before the *postfire()* methods;
- *postfire()*: last method called during an iteration;
- *wrapup():* called once at the end of execution;

The Executable interface is implemented by two different classes: the Director and the AtomicActor. The Director is responsible for controlling the execution. Typically, a Director contains a scheduler and information shared[3] between components, such as a model of time. A Director can be associated with only a hierarchic actor. Different hierarchical actors might be associated with different directors.

The AtomicActor class is a specialization of the Entity class, and provides generic implementations for the execution methods. A designer implements the behavior of a component by overriding some execution methods of the AtomicActor class. The following code shows a *fire()* method that computes the sum of the data available on all channels of a input port:

2 One level of hierarchy is also called a topology.

3 There are no shared variables in Ptolemy.

```
fire() {
    Token sum = new Token();
    for(int i = 0;i < input.getWidth();i++) {
        if(input.hasToken(i)) {
            Token t = input.get(i);
            sum = sum.add(t);
        }
    }
    output.broadcast(sum);
}
```

The *getWidth()* method of a port returns its number of channels. A Token is an object that stores a data to be exchanged. The Ptolemy II framework provides several different types of Tokens, carrying different data types. The above code operates on all possible values, since it uses a polymorphic method for addition.

In order to implement a MoC in Ptolemy II, one has to specialize the Director class and implement the Receiver interface.

3 Related Work

In [LEE02], a technique is presented for determining if a component is compatible with a given MoC. Their approach is based on extending the concept of a type system. An automata based formalism is employed to model the calls to methods of the Ptolemy API by a component,a Director, and a Receiver. Different automatons may be constructed depending on desired the level of detail. Compatibility is then checked by composing the obtained automatons.

Their work share some similarity with ours, since we also check the validity of a component by looking at the calls to the Ptolemy API. However, when possible, we try to precisely identify a component as valid or not to a MoC. For instance, a valid SDF component requires a constant number of data being exchanged. It is not clear if it is possible to model this type of information using the automata formalism. Also, we deal with whole specifications, i.e, atomic components and hierarchic components, whether in [LEE02] only atomic components are addressed. It is mentioned that it is possible to use the same technique for checking proprieties of a specification, but no detail is given.

4 MoC Validation

Our objective is given an executable specification[4] written for the Ptolemy II framework, determine what MoC are valid for it. A valid specification is one that respects the restrictions imposed by a MoC[5]. A specification may violate the rules of a MoC at three locations: 1) within the component's source code; 2) at one level of hierarchy; 3) between levels of hierarchy. This naturally leads to a bottom-up verification of the specification.

We have developed a tool for automatically performing such validation. The input to the tool is composed of each component's source code and a file containing the interconnection information. The result produced by the tool is a set of messages to the designer and possibly a modified version of the input specification. The generated messages indicates warnings and errors in the specification relative to a MoC. Figure 1 presents the main subtasks of the validation process.

4 We are considering only static and untimed specifications.

5 The restrictions come from the MoC semantics and from Ptolemy's implementation of the MoC.

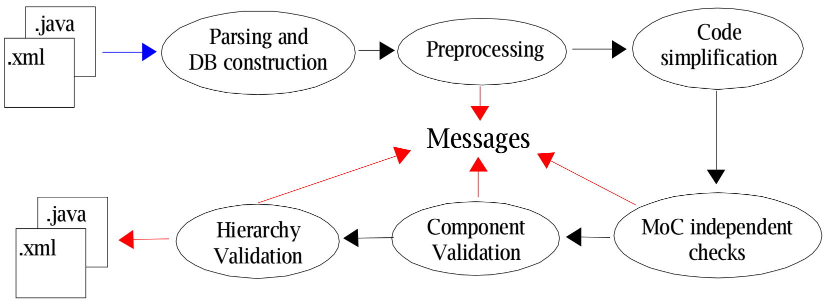

Figure 1 – The subtasks of the MoC validation.

The first subtask parses the input data (Java classes for components and a XML[BHA02] file for the interconnection information) and constructs the internal data representations. For hierarchic components, a list of contained atomic and hierarchic components is created. For atomic components, data structures [APP98] such as control-flow graph (CFG) and symbol table are generated.

Once the internal representation is created, a preprocessing phase is performed. It is divided into two steps: 1)determination if an atomic component is specifically written to a MoC; 2) identifying if the designer assigned a component as valid or invalid for a MoC. The first step searches the component's code for the presence of methods and constructs specific to a MoC. For example, a special type of atomic component of the SR model is specified by the presence of a marker attribute. The second looks for the presence of two attributes. One identifies the component as valid for a set of MoC and the other as invalid. At the end of this subtask, the components are classified as generic or specific to a MoC. A list of components to be further analyzed is constructed.

The third subtask is the simplification of the internal representation of the atomic components. This is necessary in order to present a more precise data to the following subtasks. Currently, two algorithms are employed: constant propagation and dead code elimination. These algorithms run on the static single analysis form (SSA) [APP98] of each CFG. For some MoC, it is possible to perform additional substitutions. For instance, the *hasToken()* method under SDF equals to the boolean value **true**. These MoC specific substitutions are done right before the simplification subtask.

After simplification is completed, each component has two (one for SDF, one for SR) simplified SSA CFG for each of its methods. For every simplified CFG, a new graph is generated, called Method Graph (MG). The nodes of a MG are method call nodes of the CFG to the Ptolemy II API. There is an edge between two nodes in the MG if there is a path between the respective nodes in the CFG. Auxiliary nodes are used in the MG to indicate a loop beginning and a call to another method of the source code. All analysis of atomic components described in sections 5 and 6 are based on method graphs. Figure 2 shows the MG under SDF and SR models for the *fire()* method depicted in section 2.

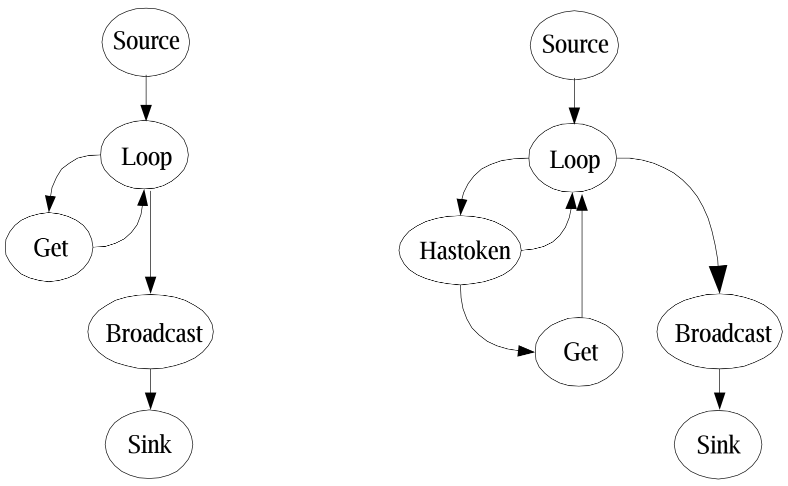

Figure 2 – Method graphs under SDF (left hand side) and SR (right hand side).

After the simplification subtask, three restrictions to the source code are checked. These restrictions are imposed by our tool in order to be able to analyze the source code. They are:

- data exchange methods can be employed only within countable loops;
- the field of a data exchange method specifying the port's channel should be a constant;
- when a vector of data is transmitted between components, its length should be specified by a constant.

If all the above restrictions are met, the MoC validation algorithms are executed.

5 Synchronous Data Flow

The synchronous data flow (SDF)[LEE87] is an extension to the data flow model of computation. In this model, the system is considered to be a graph where a node represents the function being computed by a component, and an edge indicates the data dependency between functions. The execution of each component is divided into a series of atomic activations. A component is activated when data is available for all its inputs. In the SDF model, integer values (sample rates) are used to specify the amount of data required by each input, and the amount of data produced by each output at the end of one activation. Conceptually, a well defined SDF specification should be executed on infinite input streams without termination. It is show that a SDF specification possesses a static scheduling, i.e, the number of times each component should be activated, and the order of components activation can be determined before execution. In Ptolemy II, a component activation calls the *prefire()*, *fire()* and *postfire()* methods in that order.

Following our strategy for validating a specification, we have to ensure that three conditions are met:

1. A component must produce and consume a constant number of data;
2. The sample rates of all components must be consistent [LEE87];
3. The specification must be deadlock free.

Our tool checks the first condition by applying an algorithm that computes the minimum and maximum values for the sample rates of each component's port. The algorithm traverses the method graph of the the *initialize()*, *prefire()*, *fire()* and *postfire()* methods. For each node, the values are computed based on the values of all its inputs nodes, and the method being called by the node. When encountering a graph link node, the algorithm is applied to the method graph associated to that node. When a loop head node is found, the algorithm first processes the loop body. The result is obtained at the sink node of the graph. For a constant sample rate, the minimum and maximum values have to be equal.

This and other algorithms employed in our tool are based on traversing the paths of method graphs. By doing so, we consider that all paths are executable. Since the MG is generated from a CFG, there might be false paths. It is easy to show that by applying an algorithm that tries to remove false paths from the CFG, the result of the tool can only be improved. This means that a component detected as SDF valid, will not became invalid, or with altered sample rates. What might happen is that a component once detected as non valid by the tool, is discovered as valid after the removal of some false paths.

We check the second and third conditions if all components have constant sample rates. This is done by trying to compute a schedule for the topology. If the sample rates are not consistent, it will not be possible to compute the number of required activations of each actor. If there is a deadlock situation[6], the scheduler will fail to compute the order of activations. Both faults are reported by the scheduler. We use the available SDF scheduler of the Ptolemy II framework.

6 Synchronous Reactive

The synchronous reactive (SR) model [EDW97] is based on the synchrony hypothesis: given a set of external stimulus, the system can compute an answer infinitely fast. Each reaction of the system is instantaneous and atomic. Time is divided into a series of discrete instants.

The Ptolemy's semantics for the SR model treats the components and their interconnections as a system of equations. At each instant, one has to find the least fixed point solution to this system. In order to guarantee the existence of such a solution and ensure a deterministic execution, two characteristics are imposed: 1) a signal should carry a flat[7] pointed complete partial order (CPO); 2) and a component should compute a monotonic function.

The first condition means that a signal has two states: undefined and defined. At the beginning of each instance, except for some system inputs, all signals are at the undefined state. A defined state is one were the value, or the absence of a value, is known. The second condition restricts a function to once an output signal is set to the defined state, it can not return to an undefined state or has its value changed, given a more defined set of inputs.

Within Ptolemy's implementation of the SR model, each instant is composed of a series of component activations. An activation is atomic and executes the *prefire()* and *fire()* methods. The order of component activation can be computed statically. At the end of one instant, the *postfire()* method of each component is executed. A component is classified into two types: strict and non strict. A strict component requires all inputs to be defined prior to activation, and a non strict component does not. Therefore, a strict component is activated at most once during an instant, whether a non strict component may be activated several times.

From the above characteristics of the SR model, we have to check two restrictions of an atomic component:

1. a components should generate at most one data element per each output during an instant;
2. a component must implement a monotonic function.

The first restriction is determined by using the algorithm that computes sample rates developed for the SDF model. Here, the acceptable values for an output are (0,0), (0,1) or (1,1). For outputs, values greater than one generate an error message, and rejects the component. For inputs, a value greater than one prompts a warning message, indicating that the same value will be read.

Determining if a component implements a monotonic function just by looking at its source code is a difficult task. This is because one has to find all conditions under which an output is defined, which may depends on the presence of signals and their values. For a non strict component, these conditions have to be determined considering a finite number of activations.

We have adopted a more restricted version of the SR model, were an output of a component requires the known state of all inputs that it depends on. A component is activated only when is pos-

6 The interconnection of components contains at least one loop without initialization values.

7 Flat means that the signals carries only scalar values. This is a restriction imposed by Ptolemy's implementation, not the semantics of the model.

sible to define some new output. We no longer classify a component as strict or non strict. Also, it is not allowed to define an output based on an unknown state of an input signal, i.e, an output definition that depends on the false value of the method *isKnown()*. It is easy to show that with such restrictions, only monotonic functions can be captured.

An output depends on an input under two circumstances:

1. the data being generated requires a known state of the input signal;
2. a conditional expression on the control flow path that leads to the definition of an output requires a known state of an input signal.

The two conditions for dependency are determined by searching use-definition chains [APP98] that were created during the simplification phase of the analysis. The following pseudo-code shows a simplified version of the algorithm that determines the dependency:

```
FindDependencies( MethodGraph MG, FlowGraph FG) {
    forall node N in MG do
        if N is an output signal definition
            forall uses USE of variable V in N do
                get definition DEF site of USE
                searchDataDependencies(DEF)
            forall expressions EXPR in the path leading to N do
                forall use USE of variable V in EXPR do
                    get definition DEF site of USE
                    searchDataDependencies(DEF)
}

searchDataDependencies(DefinitionSite DEF) {
    if DEF is a method call of an input signal IN
        add IN to the dependecy list
    else
    if DEF is a method call MC
        set MG the Method Graph of MC
        set FG the Flow Graph of MC
        FindDependencies(MC, FG)
    else
        forall use of variable V in the definition statement DEF
            get definition DEF' site of V
            searchDataDependencies(DEF')
}
```

Once all components are validated, checking the topology is done by trying to compute a schedule. A graph representing the dependencies between input and output signals is created. A node in this graph is a component's port. There is an edge between two nodes when: 1) both nodes belong to the same component, and there is a input/output dependency; 2) the nodes belong to different components, and there is a link in the topology connecting them. A valid SR topology is one were the obtained graph is acyclic. A topological order of the graph gives a schedule of activation.

7 Heterogeneous specifications

The validation process is performed bottom-up. First, all atomic components are analyzed. Then, each topology is validated, starting from the leaf ones, i.e., those that do not contain any hierarchic component. Once a hierarchic component is validated for a group of MoCs, it will be seen as an atomic component at the next level of hierarchy.

For each hierarchic component contained in a topology, we have to validate their interaction. For the SDF and SR models, we have:

- A SR hierarchic component contained in an SDF topology: the SR component must be homogeneous, i.e., all sample rates of its ports must be equal to 1.
- A SR hierarchic component contained in a SR topology: all output ports of the SR component are considered dependent on all its inputs.
- A SDF hierarchic component contained in an SR topology: the SDF component must be homogeneous and all output ports of the SDF component are considered dependent on all its inputs.
- A SDF hierarchic component contained in an SDF topology: the hierarchic component will be scheduled with the computed sample rates. These rates are obtained from the connected ports on the inside topology.

8 Experimental results

The first experiment we have conducted tested the analysis of atomic components. The Ptolemy II framework comes with libraries of such components. Most of them are designed not to a particular model of computation[8], i.e., they are written only with methods from the abstract semantics layer. We obtained the following results:

Library	*Total N° of Components*	*Success*	*Unable*	*Not possible*
Array	6	2	2	2
Conversion	17	16	1	0
Flow Control	16	3	3	10
Logic	5	1	1	3
Math	16	9	6	1
Random	4	3	0	1
Signal Processing	16	8	7	1
Sink	12	12	0	0
Source	11	8	1	2

8 This is called a domain polymorphic component [BHA02].

The first column specifies a component library from Ptolemy II. The third column accounts for the number of components correctly analyzed. The forth column indicates the number of components that the tool was unable to analyze, or produce the correct output, due to its limitations. The last column shows the number of components that are not possible to analyze.

For each component that the tool was unable to produce the correct result, we have inspected the reason. There were only two different situations:

1. the consumption/production was dependent on a component's parameter value;
2. the constant propagation algorithm was not powerful enough.

These two situations can be fixed by a simple improvement of the tool. Currently, we do not analyze parameters. These values can be considered as constants.

Looking at the components that were impossible to analyzed, we have identified three different situations:

1. the component's source code does not follow the restrictions imposed by our tool;
2. exceptions were being throw at the component's execution methods;
3. the component's ports are specified by the user as parameters.

The first situation can be addressed by using more powerful algorithms that analyze the source code, trying to identify the possible values of a variable. The second and third cases can only be removed by rewriting the source code of the component.

The second experiment is the analysis of a hierarchic specification. We have selected a Ptolemy's demo available for the SR model, that models a cyclic token-ring arbiter. Figure 3 shows the topology of the specification.

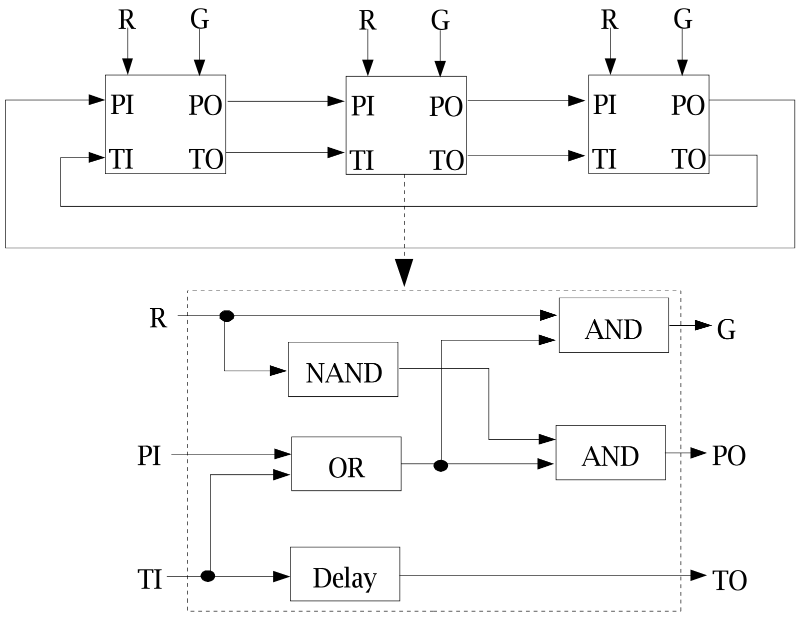

Figure 3 – Cyclic token-ring arbiter.

The specification contains two levels of hierarchy. Three leaf hierarchic components are connected at the top level. Each leaf hierarchic component contains four components computing logic functions, and a component implementing a delay. The delay component produces the value from the previous iteration.

The version available from Ptolemy II is designed for the SR model. Each logic function is implemented by a non strict component. For instance, a logic OR function can define its output as soon as one input has a known value equals to true. Under our restriction of the SR model, this is not possible. All logic function components will require the known state of all input signals.

All the atomic components were correctly identified as valid, for both SDF and SR. The leaf hierarchic component was also identified as valid for both MoCs. The top level hierarchic component was identified as invalid. This is because the presence of a cycle that goes through the PI and PO signals. The original specification did not have this problem, since the logic function components were non strict. In order to fix this problem, an extra delay actor has to be inserted before a PI signal. Its initial value should be set to true. After this modification, the tool correctly identify the top level hierarchic component as valid for both models.

9 Conclusions and Future work

In this paper, we have described a tool that given an executable specification, tries to identify the valid models of computation for it. This is done by the static analysis of the specification's code. This strategy has a problem that it is not always possible to produce a correct answer, since it is based on a static analysis of source code. Hence, our algorithms were designed to be conservative.

Despite the intrinsic difficulty of the problem, our initial results indicates the usefulness of the tool. Out of 103 components, 62 were correctly analyzed. This number can be easily risen to 83 components. We have also successfully analyzed some toy specifications.

The most immediate task that remains to be done is the improvement of the implementation of the tool, considering the situations described in section 8. Also, the analysis of specifications of actual systems will be performed.

Another important future work is the use of parameters indicating semantic characteristics of a component or signal, without implying a particular model of computation. This will prune the possible candidate MoCs.

10 References

[APP98] A. W. Appel, **Modern Compiler Implementation in ML**, Cambridge University Press, ISBN 0-521-58274-1, 1998.

[BHA02] S. S. Bhattacharryya, et al, **Heterogeneous Concurrent Modeling and Design in Java**, Memorandum UCB/ERL M02/23, University of California, Berkeley, August, 2002.

[EDW97] S. A. Edwards, **The Specification and Execution of Synchronous Reactive Systems**, PhD Dissertation, Report ERL, No. M97/31, Dept. EECS, University of California, Berkeley, 1997.

[LEE87] E. A. Lee, D. G. Messerschmitt, **Synchronous Data Flow**, Proceedings of the IEEE, Vol. 75, No. 9, September, 1987.

[LEE02] E. A. Lee, Y. Xiong, **Behavioral Types for Component-Based Design**, Memorandum UCB/ERL M02/29, University of California, Berkeley, September, 2002.