Translation of Heterogeneous High-level Models to Lower Level Design Languages

Brian A. Jackson

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Electrical and Computer Engineering

APPROVED:

Dr. James R. Armstrong, Chairman

Dr. Ezra Brown

Dr. F. Gail Gray

Dr. Dong S. Ha

Dr. Wayne A. Scales

April 2005
Blacksburg, Virginia

Keywords: SystemC, Ptolemy, Perl, XML, ASIC, FPGA, Top-down / Bottom-up Design

Translation of Heterogeneous High-level Models to Lower Level Design Languages

Brian A. Jackson

(Abstract)

Proceeding from a specification, one develops an abstract mathematical model of a system, or portion of a system. This model of a system is validated to insure that the specification is interpreted accurately and to explore different algorithms for implementing the system behavior. We use the words "portion of a system," because only rarely are systems designed wholly using a purely top-down approach. Commonly, the design approach is a mixture of top-down and bottom-up. But even in this mixed approach, top-down techniques are critical to the development of new, advanced system features and improving the performance of existing system components. An example of this style of design tools and environments is Ptolemy II. Ptolemy II is a high-level modeling tool created at UC-Berkeley. It supports heterogeneous and homogeneous modeling, simulation, and design of concurrent systems. High-level modeling of such embedded systems as digital electronics, hardware, and software can be effectively represented.

The bottom-up design approach exploits design reuse to achieve the productivity necessary to build complex systems. Historically, chip design companies have always reused designs in going from one product generation to another, but the efficiency of bottom-up design is enhanced by the use of IP (Intellectual Property) cores that a company can buy from an outside source. Design libraries are useful for system design and are an example of IP cores.

A sound methodology to translate Ptolemy models to SystemC models would have a very beneficial effect on the CAD/EDA industry. Ptolemy II is written in Java and its high-level designs, or abstract graph models, are represented as XML documents. Ptolemy's major emphasis is on the methodology for defining and producing embedded software together with the system in which it is embedded. SystemC is written in C++, and its industrial use is gaining momentum due to its ability to represent functionality, communication, software, and hardware at various levels of abstraction. SystemC produces synthesizable code. A methodology to convert Ptolemy models to synthesizable SystemC code would be the technical epitome of a hybrid between top-down and bottom-up design styles and methodologies. Such a methodology would enable system designers to obtain fast design exploration, efficient IP-reuse, and validation. Ptolemy has various components and models of computation. A model of computation dictates how components

interact between other components. SystemC has its own models of computation and design libraries. XML and Perl are both powerful tools by themselves, and we use these tools in this research to create a sound methodology for translating Ptolemy models (high-level of abstraction) to synthesizable SystemC code (low-level of abstraction), i.e.: code which can serve as input to hardware tools.

Acknowledgments

First, I would like to thank God for maintaining my personal perseverance of seeing this hard work through. Next, I thank my Lord and Savior Jesus Christ for doing things for me that my words alone can not begin to express. I thank Him for allowing me to grow through tough spiritual and emotional tribulations, and constantly reminding me that He would never leave me.

Next, I would like to thank a plethora of people who helped me along the way in different senses. In the academic sense, I thank my advisor, Dr. Armstrong, who presented a real challenge to me in terms of learning; applying my own intuition; growing intellectually; keeping me on point research-wise; and guiding me through the particulars of this research. I thank my committee members, Drs. Gray, Ha, Scales, and Brown for their invaluable patience and their valuable input into my research. I thank my unofficial "committee members" at Lucent Technologies in Whippany, New Jersey, for answering real world, industry questions in regards to my research.

In the non-academic sense, I thank my advisor again, Dr. Armstrong, for the short random chats we had during office and non-office hours. I also thank him for motivating me from time to time when my confidence in myself was shaken. But, most importantly, I thank him for a friendship that's grown over the years since being my advisor during my masters program. That means a lot to me. I thank the friends I made while an undergraduate at The University of North Carolina at Charlotte and Johnson C. Smith University. I thank the friends I made here at Virginia Tech. I thank my former manager at Lucent Technologies (when it was formerly call AT&T Bell Laboratories), Janet L. Benton, for planting that initial seed inside my head to go beyond the masters degree and pursue a doctoral degree. Her friendship and personality was (and still is) inspiring to me. I still consider her a friend. I thank those "old school" teachers when I was growing up in Brooklyn, New York. Their firm and disciplining teaching styles helped mold me into a person who now looks back and can appreciate them for all they have done for me. This is especially true of Mrs. Sarah J. Barrington during my days at and after Epiphany

Lutheran Elementary School. I thank Dr. Peter Hall (Johnson C. Smith University) and Dr. Silvia J. Middleton (UNC-Charlotte) when they were my academic advisors during my undergraduate years. Their faith and support in me is something I will never forget. During the period when I went into academic probation and lost confidence and focus, they were the main ones in academia who looked at me as a person and helped me emotionally in ways I can never repay. They were instrumental in building my academic confidence. I thank my fraternity brothers for demonstrating to me time and time again that friendship is indeed essential to the soul. I thank my pastor, Pastor Albert Delmadge, for just "being there" and talking to me. His actions as a servant of God have helped my spiritual faith tremendously to the point where I have a better relationship with God and Jesus Christ. I thank all those people (mentioned and unmentioned) who called just to say hello and to ask how I was doing as person instead of asking for something. That selfless act helped my soul in ways I can not begin to convey.

I want to thank God for allowing me the chance to see my grandmother, Ida Julia King, before she passed away and entered His Kingdom. I bless the days we got reconnected after many years of separation. "Mommy King" will always have a special place in my heart and I will never forget her. I think about her almost daily. I wish she were here but I know that God has her in a better place and He allows her to see how I'm doing from time to time. I still miss her. Her words of encouragement when she was still on this world are one of the many sources of strength for me. I sincerely hope that one day I will become the great man that she saw in me. Thank you, Lord God, for allowing me that chance to see her before she entered Your Kingdom.

Lastly, I thank my parents, Aubrey and Annette Jackson, who offered words of encouragement and supported me mentally and emotionally. Their being there for me has put me where I am today. To Daddy, your hardworking ethic has rubbed off on me. Your actions alone showed me to never give up and keep on trying no matter what or who stands in your way. Even with that constant pressure, you still made time for me when I was growing and showed me love and discipline. Your spending time with me showed

me that there are things more important than making money. I wish I can someday become half the man you are. You’re my hero.

To all those mentioned and unmentioned, I dedicate this research to you all. This is rightfully yours as well as it is mine. I didn’t do this all by myself. Your prayers, the talks, the fellowship, the encouragements, the kind words, and hugs kept me going. I thank you all especially for the prayers. My knowing that I am important to each of you enough to the point where you send up prayers touches my heart, mind, and soul. I mean it. I can honestly say that all this hard work would mean absolutely nothing to me if I couldn’t share this with those significant people in my life. As God as my witness, I sincerely mean that from the utmost bottom of my heart in all humility. I love you. Thank you all.

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

The use of high-level modeling tools has made a significant contribution to the CAD (Computer-Aided Design) industry in terms of abstract model representation. The ability to represent an abstract model graphically allows designers to quickly realize a system without having to deal with restrictions such as maximum clock frequency, path delays, and design area to name a few. High-level modeling tools thus provide a validation and verification capability that allows designers the required level of confidence to proceed to the next level in the design process. Figure 1.1 shows a representation of the design process. Level 1 represents the ideal abstract model. This model is usually the initial representation of the written design specifications. Level 2 typically represents a behavioral description of the proposed system using a lower-level modeling tool that employs a hardware description or system-level design language. The languages can form the input for synthesis tools which allows one to proceed to lower (>2) levels. The goal of proceeding down the design process is to reach Level *n* which represents a physical implementation.

1.1 Top-down Design Methodology

Proceeding from a specification, one develops an abstract mathematical model of the system, or portion of a system. This model of a system is validated to insure that the specification is interpreted accurately and to explore different algorithms for implementing the system behavior. We use the words “portion of a system,” because only rarely are systems designed wholly using a purely top-down approach. Commonly, the design approach is a mixture of top-down and bottom-up. But even in this mixed approach, top-down techniques are critical to the development of new, advanced system features and improving the performance of existing system components. This design methodology is typically implemented with high-level modeling tools. These modeling tools provide designers the ability to rapidly prototype a system through a GUI-based, specification-oriented approach.

1.2 Bottom-up Design Methodology

This approach exploits design reuse to achieve the productivity necessary to build complex systems. Historically, chip design companies have always reused designs in going from one product generation to another, but the efficiency of bottom-up design is enhanced by the use of IP

(Intellectual Property) cores that a company can buy from an outside source. The challenges with using these outside cores is designing interfaces between the cores and other logic and being confident of their correctness. Virtual Socket Interface Alliance (VSIA) [1] has been trying to standardize IP core interfaces, validation data, and other kinds of data to be delivered with cores to make this style of design easier for the industry. An example of a bottom-up design methodology is the use of either an HDL (hardware description language) or a system-level design language. Either of these two allows designers the ability to take on a bottom-up, IP-reuse centric approach.

1.3 High-level to Low-level Model Translation

To date, there is no automated process to translate high-level models into a form where models can be combined with lower level core models. In this work, we will use high-level models created with Ptolemy II. Most Ptolemy models have gone into software since its major emphasis is on the methodology for defining and producing embedded software together with the systems within which it is embedded [2,3]. Thus, the objective of this research is to create a methodology to go from a GUI abstract model to an HDL or system-level design language representation which targets a hardware implementation. Specifically, in this research, the proposed methodology goes through 5 levels of abstraction in the design process. The GUI abstract model is represented by Level 1. Level 1 is created using the high-level modeling tool called Ptolemy II, which handles abstract models in both heterogeneous and homogeneous modeling domains [2]. Ptolemy is written in Java and was created at UC-Berkeley. Its behavioral models are Java coded actors, and its structural models are entered graphically and represented as XML documents. Ptolemy is widely used in academia and commercial versions have begun to emerge. Levels 2 through 4 are represented in SystemC. SystemC is a set of C++ classes for system-level modeling that can be used for hardware software co-design [4]. It produces synthesizable code and enjoys wide industrial support [5]. In this research, Level 2 represents the initial SystemC model while the remaining lower levels (>2) in the design process, with the exception of Level 5, represent refinements and modifications to this initial model. These SystemC refinements and modifications consist of making the SystemC model synthesis-ready. To do so, the register-transfer level (RTL) model is the modeling style [4] used to represent the lower levels. We will further discuss the nature of the refinements in Levels 2 through 4 in Chapter 2. Level 4 is regarded as a properly styled SystemC model. This properly styled code is synthesis-ready and is used as input to a synthesis tool of a designer's choosing. The output of this synthesis tool produces a specific coded style for either an ASIC (Application-Specific Integrated Circuit) or an FPGA (Field Programmable Gate-Array) physical implementation. This physical implementation is represented by Level 5.

Background knowledge of high-level modeling tools, hardware synthesis, and system-level and hardware description languages are essential for developing our top-down methodology. A basic understanding of both Java and C++ is also needed. Knowledge of XML (Extensible Markup Language) and Perl (Practical Extraction and Reporting Language) are also important as they are used to implement the model refinement steps.

1.4 Dissertation Outline and Results

A sound methodology to translate Ptolemy models (high level of abstraction) to SystemC models (low level of abstraction) would have a very beneficial effect on the CAD/EDA (Electronic Design Automation) industry. In this research, a transformation methodology is defined and implemented using XML and Perl scripts. Chapter 2 discusses the models of computation (MOCs) of Ptolemy and SystemC. Chapter 3 describes the mathematical models used to represent the translation between Ptolemy and SystemC models when proceeding down the 5-level design hierarchy. Chapter 4 shows the XML structures used to translate Ptolemy actors to SystemC modules and gives the rationale for using XML. Chapter 5 outlines the transformation methodology to step-by-step translate a Ptolemy model to a synthesis-ready SystemC model. Perl scripts are presented to implement the translations. Chapter 6 demonstrates the effectiveness of the transformation methodology by showing how several heterogeneous Ptolemy models that utilize various models of computation can be translated to a synthesizable description and then synthesized to a gate level circuit. Finally, Chapter 7 summarizes the results and transformation methodology, draws conclusions, and suggests future work.

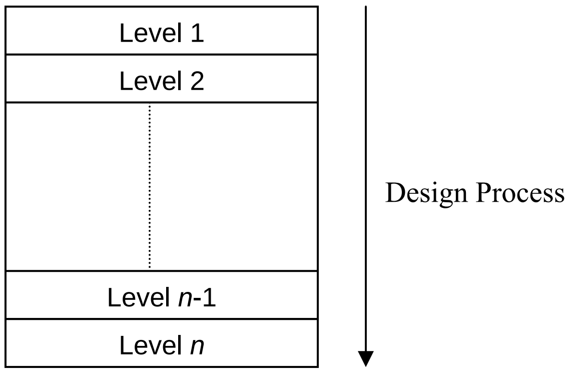


Figure 1.1. Levels of abstraction of the design process.

Chapter 2. Models of Computation of Ptolemy and SystemC

A model of computation is defined as a domain that dictates execution and communication methods in a modeling environment. In Ptolemy, a *director* dictates the model of computation (MOC) to be used. Each MOC falls under the category of either a timed or untimed domain. In a timed domain, events (or computations) are executed on a timed scale while, in an untimed domain, events are merely ordered. There is no specific notion of time [2]. Table 2.1 shows a list of some of the MOCs implemented in Ptolemy.

Table 2.1. Ptolemy Domains

Timed	Untimed
Discrete Event (DE)	Process Network (PN)
Hardware Discrete Event (HDE)	Synchronous Data Flow (SDF)
Distributed Discrete Event (DDE)	Finite State Machine (FSM)
Discrete Time (DT)	Communicating Sequential Process (CSP)
Continuous Time (CT)	Synchronous Reactive (SR)

As stated above, an MOC has a method of execution and a method of communication. In SystemC, the method of execution is the discrete event (DE). The method of communication is the channel which can be implemented at varying levels of abstraction. FIFOs provide the high level method of communication while signals provide the low level of communication. Two important SystemC MOCs that utilize these two levels of communication are the Kahn process network (KPN) and the register-transfer level (RTL) MOCs. The KPN MOC utilizes FIFOs and is effective for modeling the initial representation of systems and is widely used for signal-processing applications [4,6]. The RTL MOC utilizes signals and is effective for modeling digital hardware synchronized by clock signals. In this research, to implement the KPN MOC, SystemC *sc_fifo* channels and SC_THREAD processes are used. To implement the RTL MOC, SystemC *sc_signal* channels and SC_METHOD processes are used.

As stated in the preceding paragraph, the method of execution for SystemC is the discrete event (DE). SystemC deals only with discrete-event MOCs. As a result, continuous time models cannot be constructed in SystemC. This limits the types of Ptolemy directors that are acceptable in this research for Ptolemy to SystemC translation. For example, the Continuous Time (CT) director would not fit the 5-level design process outlined in Chapter 1 in regards to a SystemC method of communication. Table 2.2 shows the 5-level design process of the methods of communication used for the Ptolemy to SystemC translation.

Table 2.2. Methods of Communication of Design Hierarchy

Hierarchy Level	Model	Method of Communication
Level 1	Ptolemy	Token Dataflow Mechanism
Level 2	SystemC	sc_fifo<T>
Level 3	SystemC	sc_signal<T>
Level 4	SystemC	sc_signal< sc_int<W> >
Level 5	Synthesized ASIC/FPGA	

Section 2.1.1 discusses the Ptolemy token dataflow mechanism in Level 1. The variable *T* in Levels 2 and 3 represents a SystemC data type (i.e.: Boolean, integer, etc.). In Level 4, *sc_int<W>* is known as a fixed-precision integral type. This type is parameterizable by the designer, or design team, with *W* denoting the width of the type in bits [4]. Thus, for example, *sc_int<4>* is a 4-bit, signed type ranging from -8 to 7. There is another fixed-precision integral type named *sc_uint<W>*. This type has the same properties of *sc_int<W>* with the exception that this integral type deals with unsigned numbers. Both these integral types have synthesis benefits in terms of resource allocation, design area, and speed. These synthesis benefits are discussed in section 2.2.1.

2.1 Dataflow inside Ptolemy and SystemC Models

2.1.1 Level 1 to Level 2 Translation

In Ptolemy, components are called *actors*. These actors execute and communicate with other actors via *tokens* [2,7]. This is the prescribed dataflow mechanism of Ptolemy. To demonstrate this dataflow mechanism, Figure 2.1 illustrates a high-level abstract representation of the execution and communication methods of a Ptolemy model. The choice of director dictates the model of computation to be used. The MOC provides the execution semantics for the actor components of the model and the communication method between actors. The model of computation is implemented in the actor receiver, which is specific to the MOC. In Ptolemy, communication between actors is performed through interconnects called *relations* [2]. As shown in Figure 2.1, the director (MOC) is responsible for dictating when actors are to execute and do their prescribed function and when data is transferred between actors via relations.

Figures 2.2a through 2.2c illustrate the execution/communication cycle of a Ptolemy model. Figure 2.2a shows the director, which is associated with a specified MOC, invoke Actor1 to do its prescribed function for the execution cycle. Once the actor completes its function, it produces a token. The token represents the actor's output data. Figure 2.2b shows the director implement the communication cycle by transferring the token from Actor1 to the receiver of Actor2. Figure 2.2c shows the director cause Actor2 to execute by processing the token at its receiver and do its actor-specific function. When Actor2 processes the token at its receiver, the token is said to be consumed and removed from the receiver. The token data is not usually retained in the receiver. Of the ten MOCs listed in Table 2.1, the HDE domain is the only director that retains token data in the actor's receiver after token consumption during the execute cycle.

As stated in Chapter 1, Level 1 in the design process is represented as a Ptolemy model while Level 2 is represented as the initial SystemC model. The SystemC version of a Ptolemy actor is called a module. When translating Level 1 to Level 2, the semantics of the Ptolemy token dataflow mechanism is preserved onto the SystemC model. Through the KPN MOC, this semantics preservation is done through the use of the SystemC constructs *sc_fifo* and SC_THREAD. Specifically, when using *sc_fifo* channels, the token dataflow mechanism is achieved through FIFOs with their blocking *read()* and *write()* methods [4]. To implement these two methods, the thread process SC_THREAD is required. With the SC_THREAD process type, synchronization is implicit [4] and no token gets lost and threaded processes are stopped and resumed automatically.

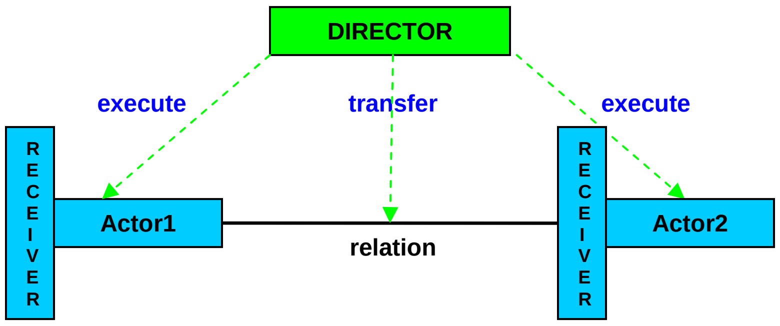


Figure 2.1. Ptolemy execution and communication model

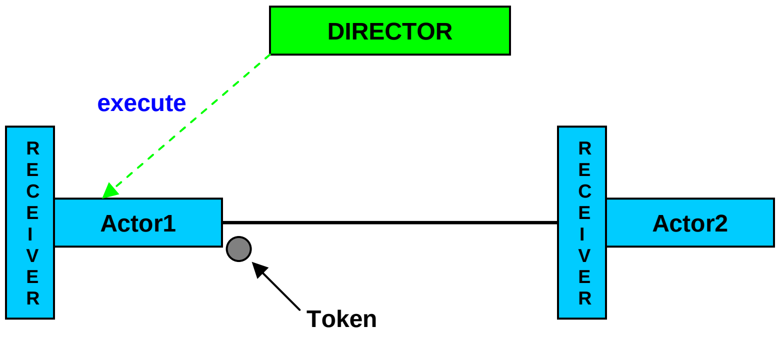


Figure 2.2a. Ptolemy actor execution stage for token production

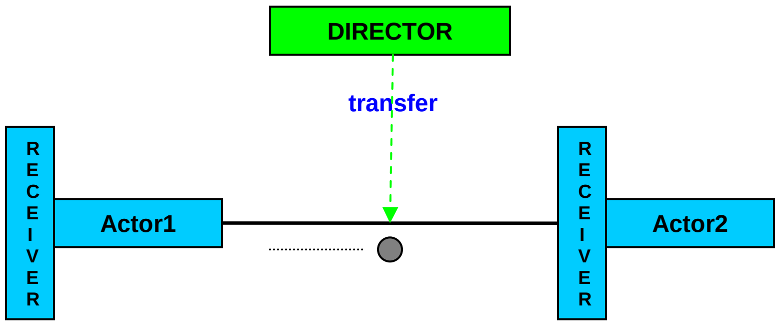


Figure 2.2b. Token communication stage between actors

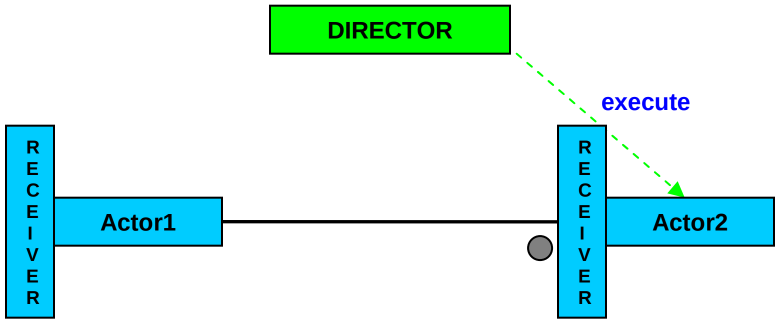


Figure 2.2c. Token consumption and actor execution stage

2.1.2 Level 2 to Lower Levels (>2) Translation

The token dataflow mechanism is semantically preserved when going from Level 1 (Ptolemy model) to Level 2 (SystemC model). Using the SystemC KPN MOC, FIFOs and thread processes ensure this one-to-one dataflow transformation. To refine this Level 2 representation so that progression down the design process continues, modifications to this initial SystemC model are required. As stated in Chapter 1, the goal of the translation methodology is to produce synthesizable code written in the system-level design language SystemC. The synthesis tool of choice in this research comes from Synopsys® and is called SystemC Compiler. Synopsys® is the most widely used logic synthesis product line in the world [8]. Specifically, for this research, their synthesis software packages are used to synthesize the code. As a result of this research decision, SystemC code is constrained to meet a synthesis format that uses synthesis-acceptable constructs. Specifically, the initial SystemC model is, by default, not synthesis-ready because the use of the SC_THREAD thread process is not a synthesis-acceptable construct. SystemC has three process types. These processes are SC_METHOD, SC_THREAD, and SC_CTHREAD [9]. Of the three, SC_METHOD and SC_CTHREAD (clocked thread process) are synthesis-acceptable constructs [28].

To properly modify the initial SystemC model (Level 2 in the design process), changes to the model of computation and process type are required. Specifically, a switch from a KPN MOC to a RTL MOC is the modification needed for a Level 2 to a Level 3 transformation as shown in Table 2.2. As stated earlier in this chapter, the RTL MOC utilizes signals referred to in SystemC as *sc_signal* channels. At Level 3, the SystemC RTL MOC employs the RTL modeling style where data transfer occurs among registers. Specifically, the RTL modeling style basically consists of combinational logic between registers where the combinational logic is the data path between the registers. The data produced from the combinational logic block is synchronously transferred through a clocked register to either another or the same combinational logic block [10]. Figure 2.3 shows this RTL modeling style.

The use of SystemC constructs SC_METHOD and SC_CTHREAD allow synthesis using SystemC Compiler. But, in keeping with the RTL modeling style, the SC_METHOD process is better suited than the SC_CTHREAD (clocked thread) process. The SC_METHOD process can model either combinational logic or a register. The SC_CTHREAD process can also model both combinational logic and registers because the process is capable of suspending in the middle of its function call through the use of *wait()* statements [9]. In RTL, as shown in Figure 2.3, neither combinational logic (i.e.: adders, multipliers, etc.) nor registers suspend in the middle of its function call. When a function, or module, is defined as a SC_METHOD, the function executes from start to end without interruption when called. In other words, once the function is called, each line of its SystemC code is executed sequentially until it reaches its end. This is the reason SC_METHOD is chosen over SC_CTHREAD in this research. Also, the SC_METHOD process simulates more efficiently than the SC_CTHREAD process.

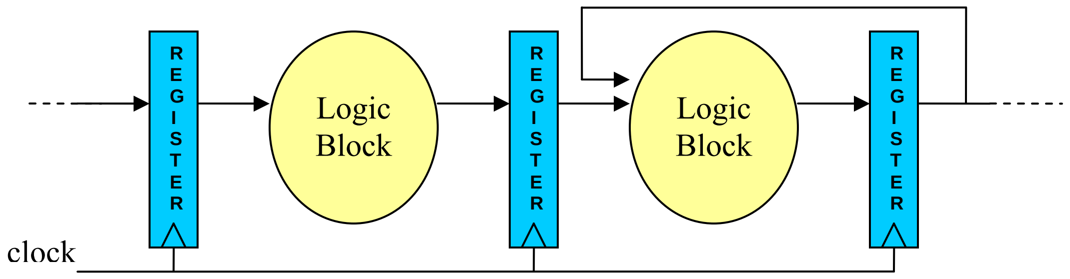


Figure 2.3. General example of RTL modeling

2.2 MOC Progression down the Design Process

2.2.1 Hardware Considerations

When implementing a Ptolemy model from written design specifications, it is the responsibility of the designer, or design team, to decide and discuss if the proposed design could have a realistic hardware, physical implementation (Level 5) either as an ASIC or FPGA. In this research, this is referred to as "designer rationale and technical common sense." This concept is a concrete guideline used in this research that states that the Ptolemy modeler must decide up front whether the model is feasible in hardware. If the model is feasible in hardware, then it could follow the 5-level design process. If the model is not hardware-feasible, then the Ptolemy modeler could either make discretionary attempts to make the model hardware-feasible or decide the model is better suited for software implementation.

When progressing down the 5-level design process, the highest level of abstraction is represented as Level 1 while the lowest level of abstraction, with the exception of Level 5, is represented as Level 4. Level 1 (Ptolemy model) encompasses multiple models of computation (MOCs) while Level 4 (synthesis-ready SystemC model) encompasses only 1 model of computation. Specifically, Level 1 could be represented by any MOC listed in Table 2.1 while Level 4 is represented solely by the RTL MOC. To demonstrate the degree of high level abstraction, Level 1 could have one or more MOCs in a Ptolemy model. Figures 2.4a and 2.4b illustrate a Ptolemy model with 2 models of computation. Figure 2.4a has a Hardware Discrete Event (HDE) MOC while the actor_module component contains a Discrete Event (DE) MOC. The contents of component actor_module are shown in Figure 2.4b. The component actor_module is referred to as a *composite actor* [2] in Ptolemy. Composite actors are hierarchical models and are themselves actors that contain actors.

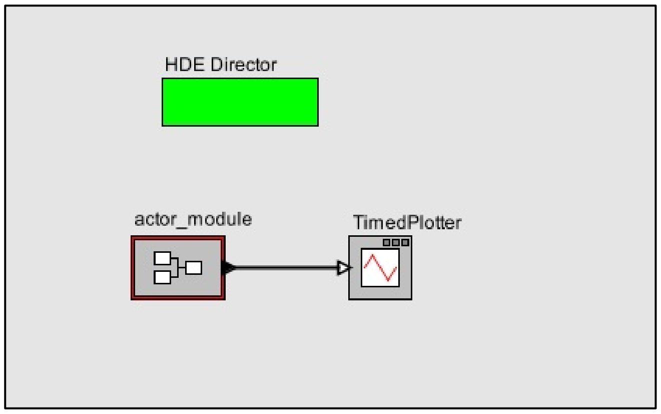


Figure 2.4a. Multiple MOC Ptolemy model

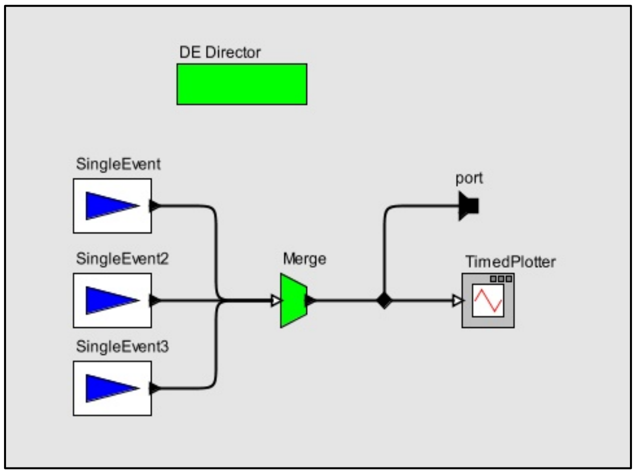


Figure 2.4b. MOC inside Ptolemy composite actor actor_module

Progressing down the 5-level design process results in a many-to-one translation in regards to Ptolemy to SystemC MOC transformation. Figure 2.5 illustrates this process. Since the goal is hardware representation, various Ptolemy MOCs will converge to a SystemC RTL MOC. In Figure 2.5, Level 1 is represented by the Ptolemy Models section (i.e.: SDF, DE, HDE, and SR); Level 2 is represented by the KPN MOC of the SystemC Models section; and Levels 3 and 4 are represented by the RTL MOC of the SystemC Models section. It is important to note that, although Level 4 is defined as the synthesis-ready SystemC model, Level 3 is also synthesis-ready. In this research, the purpose of Level 3 is for RTL simulation while the purpose of Level 4 is for RTL simulation and synthesis. The only difference between these two levels is the width of bits used when representing data. Specifically, when using SystemC Compiler, the synthesized model of Level 3 would result in an unconstrained bit width that has an extremely large range of possible values that depends on the word length of the host machine [10]. More than likely, this unconstrained bit width may be more than what is needed for that specific design use. A simple example is an *n*-bit adder circuit. If the adder is going to be used to add 4-bit operands, a synthesized, unconstrained width of 32 bits is a waste of resources whether ASIC or FPGA. The fixed-precision integral type *sc_int<W>* used in Level 4 overrides the default, unconstrained data width of SystemC Compiler, thus realizing benefits in terms of resource allocation, design area, and speed during synthesis [11].

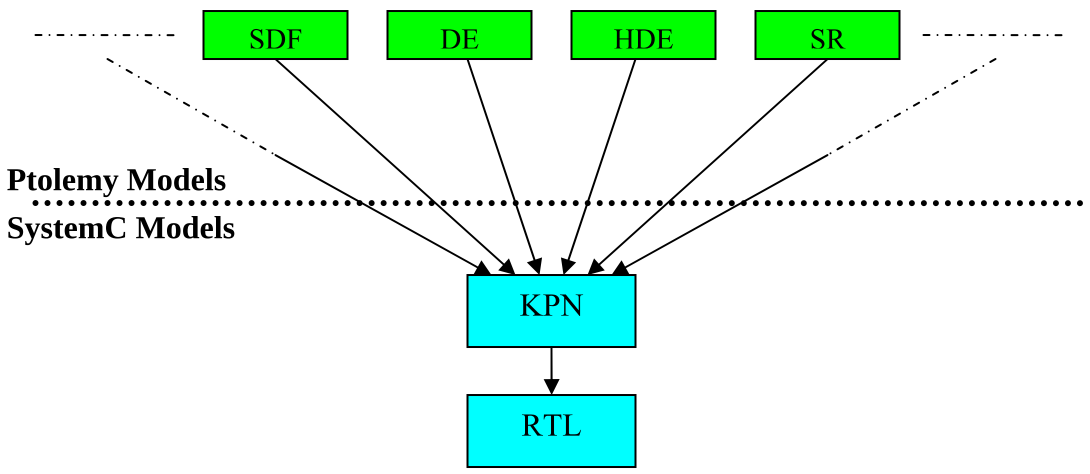


Figure 2.5. MOC transformation from Ptolemy domain to SystemC domain

Models of computation in Ptolemy are considered elaborate controllers that dictate execution and communication of its actors. Depending on the Ptolemy MOC, an actor could behave one way in a specific MOC and behave differently in a totally different MOC [2]. To design a SystemC equivalent (i.e.: an elaborate controller equivalent to a detailed finite state machine) is impractical and complex. It is impractical and complex because, when going to the lower levels of abstraction, certain mechanisms disappear such as the Ptolemy token dataflow mechanism. In the place of this particular mechanism is the SystemC *sc_signal* channel. In hardware designed using the RTL modeling style, the interconnections between modules are represented by digital signals that are synchronized by clock signals. These digital signals (assuming proper functioning and no bus contentions) will constantly have a digital value (i.e.: binary 0, binary 1).

2.2.2 Java and C++ Semantics Considerations

When proceeding down the 5-level design hierarchy, an important issue is deciding which JAVA code in an actor must be translated and which could can be ignored There are two general guidelines to follow during this translation step. Those general guidelines are as follows:

1) Critical Behavior: If a parameter and/or variable of a Ptolemy actor are such that the output port of the actor is dependent on them, then that particular line of Java code is retained and carried through;
2) Simulation or Graphic Artifact: If a parameter and/or variable of a Ptolemy actor deals solely with simulation (e.g.: length of simulation) or graphical representation, then that particular line of Java code is *not* carried through.

These guidelines are general but they strictly rely on the "designer rationale and common sense" paradigm. To illustrate these guidelines, consider the Ptolemy *Ramp* actor. This actor is composed of the parameter and variable names as follows:

1) init (parameter)
2) step (parameter)
3) firingCountLimit (parameter)
4) _stateToken (variable)

When viewing the Java code of this actor, the parameters *init* and *step* and the variable *_stateToken* meet the Critical Behavior guideline while the parameter *firingCountLimit* meet the Simulation or Graphic Artifact guideline. Therefore, *init*, *step*, and *_stateToken* are kept for translation while *firingCountLimit* is not kept for translation. For this particular actor, the parameter *firingCountLimit* deals with controlling the length of simulation when using the Ptolemy GUI name *Vergil*. This is a simulation control parameter that does not go into hardware.

Chapter 3. Mathematical Models of Ptolemy and SystemC Models

When proceeding down the various levels of the 5-level design process outlined in Chapter 2, representing the complete design model at each of the first four levels through mathematical models helps to better illustrate step-by-step the Ptolemy to SystemC transformation. The purpose of this chapter is to abstractly demonstrate nodal transformations of Ptolemy and SystemC models from level to level. The mathematical models used in this research consist of directed graphs and mappings [29,30] from level to level. Sections 3.1 through 3.3 discuss the directed graphs, or digraphs, used to model Ptolemy actors and SystemC modules, respectively. Sections 3.4 and 3.5 discuss the actual level-to-level mappings of nodes going from a Ptolemy (Level 1) domain to a SystemC (Level 2) domain and from a SystemC (Level 2) domain to a SystemC (Levels 3 and 4) domain.

In the RTL modeling sense, the goal of these mathematical models is to illustrate Ptolemy actors being mapped to an RTL circuit with combinational logic between register banks. Section 5.2 describes this in a more detailed fashion.

3.1 Level 1: Ptolemy Actor

Following [33], a Ptolemy actor schematic can be represented as a directed graph as follows:

$$D_{L1}=\{N_{L1},R_{L1}\} \qquad (3.1)$$

where N_{L1} is the set of nodes and R_{L1} is the set of relations. Let $N_{L1}=A \cup I \cup O$ where A is the set of *n* internal actors, I is the set of *m* inputs, and O is the set of *o* outputs. An internal actor is in a reference to an actor that is neither a source nor a sink actor. If an actor is not referred to as either internal, source, or sink, it is an internal actor. Each A_i (i=1…n) is a node representing a Ptolemy internal actor, I_j (j=1...m) is a source node representing a Ptolemy source actor (i.e.: a node with *in degree* of 0), and O_k (k=1...o) is a sink node representing a Ptolemy sink actor (i.e.: a node with *out degree* of 0). Each $r_i \in R_{L1}$ (i=1,2,......,q) is an arc representing a model relation. Each internal actor has *p* inputs $(i_1,i_2,\ldots,i_p)$ where the input subscripts are always a monotonically increasing set of integers. For an actor A_i, we call this input set I_{Ai}, and then define a mapping $\pi_{Ai}(I_{Ai})$ of I_{Ai} to R_{L1} which renames actor input subscripts (1,…,p) to the corresponding relation subscripts [31]. To simplify notation, we will write π_{Ai} for $\pi_{Ai}(I_{Ai})$ assuming that π_{Ai} always operates on I_{Ai}.

Figure 3.1 shows a digraph of an internal actor. Note that I={I_1,I_2}, O={O_1}, A={A_1}, R_{L1}={r_1,r_2,r_3}, I_{Ai} ={ι_1,ι_2}, π_{A1}=(1,2).

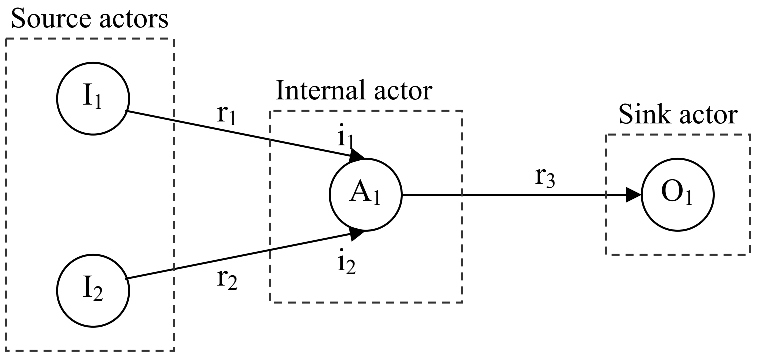


Figure 3.1. Digraph of a Ptolemy actor (Level 1)

3.2 Level 2: SystemC Module

A SystemC model, like a Ptolemy actor schematic, can be represented as a digraph as follows:

$$D_{L2}=\{N_{L2},C_{L2}\} \quad (3.2)$$

where N_{L2} is the set of nodes and C_{L2} is the set of channels. Let N_{L2}=M$\cup$I$\cup$O where M is the set of *n* internal modules, I is the set of *m* inputs, and O is the set of *o* outputs. An internal module is not a testbench component used for either producing input stimulus or displaying output responses. Each M_i (i=1…n) is a node representing a SystemC internal module, I_j (j=1...m) is a source node representing a SystemC source module (i.e.: a node with *in degree* of 0), and O_k (k=1...o) is a sink node representing a SystemC sink module (i.e.: a node with *out degree* of 0). Each $c_i \in C_{L2}$ (i=1,2,......,q) is an arc representing a model channel. Each internal module has *p* inputs (i_1,i_2,…,i_p) where the input subscripts are always a monotonically increasing set of integers. For a module M_i, we call this input set I_{Mi}, and then define a mapping $\pi_{Mi}(I_{Mi})$ of I_{Mi} to C_{L2} which renames module input subscripts (1,…,p) to the corresponding channel subscripts. To simplify notation, we will write π_{Mi} for $\pi_{Mi}(I_{Mi})$ assuming that π_{Mi} always operates on I_{Mi}. Figure 3.2 shows a digraph of an internal module. Note that I={I_1,I_2}, O={O_1}, M={M_1}, C_{L2}={c_1,c_2,c_3}, I_{M1}={i_1,i_2}, π_{M1}=(1,2).

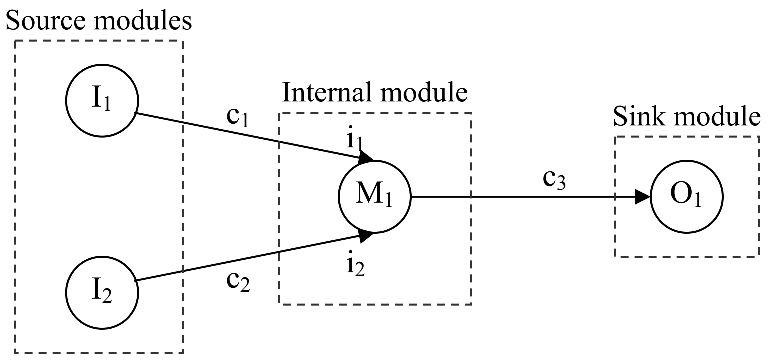


Figure 3.2. Digraph of a SystemC model (Level 2)

3.3 Levels 3 and 4: SystemC Module

For a SystemC model at these levels, the corresponding digraph is generally similar to the digraph in the preceding section but with additional nodes and channels. This digraph is represented as follows:

$$D_{L3}=\{N_{L3},C_{L3}\} \qquad (3.3)$$

where N_{L3} is the set of nodes and C_{L3} is the set of channels. Let $N_{L3}=M\cup F\cup I\cup O$ where M is the set of *n* internal modules, I is the set of *m* inputs, O is the set of *o* outputs, and F is the set of $m+o$ flip-flop modules. Each F_i (i=1…m+o) is a node representing a SystemC D-type flip-flop module used in RTL modeling. Each $c_i\in C_{L3}$ (i=1,2,......,m+o+q) is an arc representing a model channel. The variable *q* here is the same as the variable *q* used in the preceding section when defining model channels. Figure 3.3 is based off Figure 3.2 but using digraph D_{L3}. Note that $I=\{I_1,I_2\}$, $O=\{O_1\}$, $M=\{M_1\}$, $F=\{F_1,F_2,F_3\}$, $C_{L2}=\{c_1,c_2,c_3,c_4,c_5,c_6\}$, $I_{M1}=\{i_1,i_2\}$, $\pi_{M1}=(1,2)$.

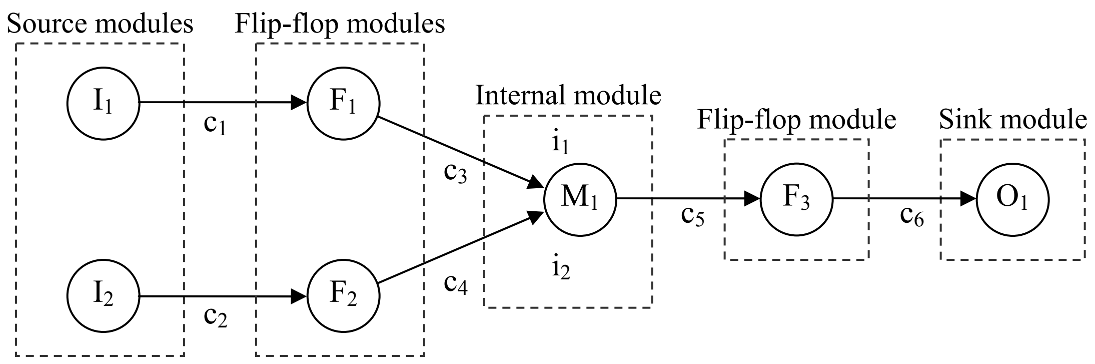


Figure 3.3. Digraph of a SystemC model under RTL modeling (Levels 3 and 4)

Equation (3.3) can represent either a cyclic or non-cyclic graph [32]. Figure 3.3 is an example of a non-cyclic graph (i.e.: *feedforward* graph). An example of either a Level 3 or Level 4 SystemC model in the form of a cyclic graph is shown in Figure 3.4. This graph represents the case where an RTL model requires feedback.

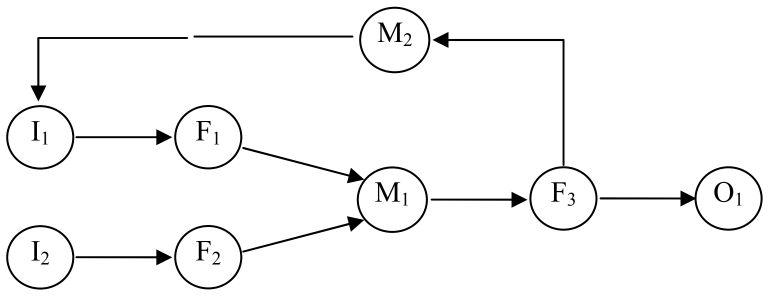


Figure 3.4. Cyclic graph to represent feedback in a SystemC model under RTL modeling

3.4 Level 1 to Level 2 Mapping

Given the definitions of digraphs of a Ptolemy actor (Level 1) and a SystemC model (Level 2), we now define how to map a Ptolemy actor digraph to a SystemC model digraph. The digraphs D_{L1} and D_{L2} represent the Ptolemy and SystemC domains respectively. Then the mapping M_{1_2} of D_{L1} onto D_{L2} is as follows:

$$\begin{aligned} &M_{1\_2}: D_{L1} \Rightarrow D_{L2} \\ &F_{N1\_2}: N_{L1} \Rightarrow N_{L2} \qquad (3.4) \\ &F_{S1\_2}: R_{L1} \Rightarrow C_{L2} \end{aligned}$$

where F_{N1_2} and F_{S1_2} are defined as follows:

1) For F_{N1_2}, there was one case considered in this research.
    a. $N_{L1} = N_{L2}$. In our implementation, this mapping is a one-to-one correspondence, or bijection [30], going from Ptolemy actor to SystemC module. Therefore, F_{N1_2} is an isomorphism. Figure 3.5 illustrates this bijective mapping.
2) For F_{S1_2}, a relation *r* and a channel *c* are both considered to have two properties: 1) connectivity and 2) storage/update mechanism. Thus, for a relation *r*, there are two mapping functions as follows:
    a. F_{Sc1_2}: Performs a bijection, or bijective mapping, of all relations in D_{L1} to channels to D_{L2}.
    b. F_{Ssu1_2}: The storage/update mechanism for each relation in D_{L1} must be mapped to a channel storage/update mechanism in D_{L2}. Specifically, a Ptolemy model has a token dataflow mechanism while, for a SystemC model at Level 2, a FIFO dataflow mechanism is also used.

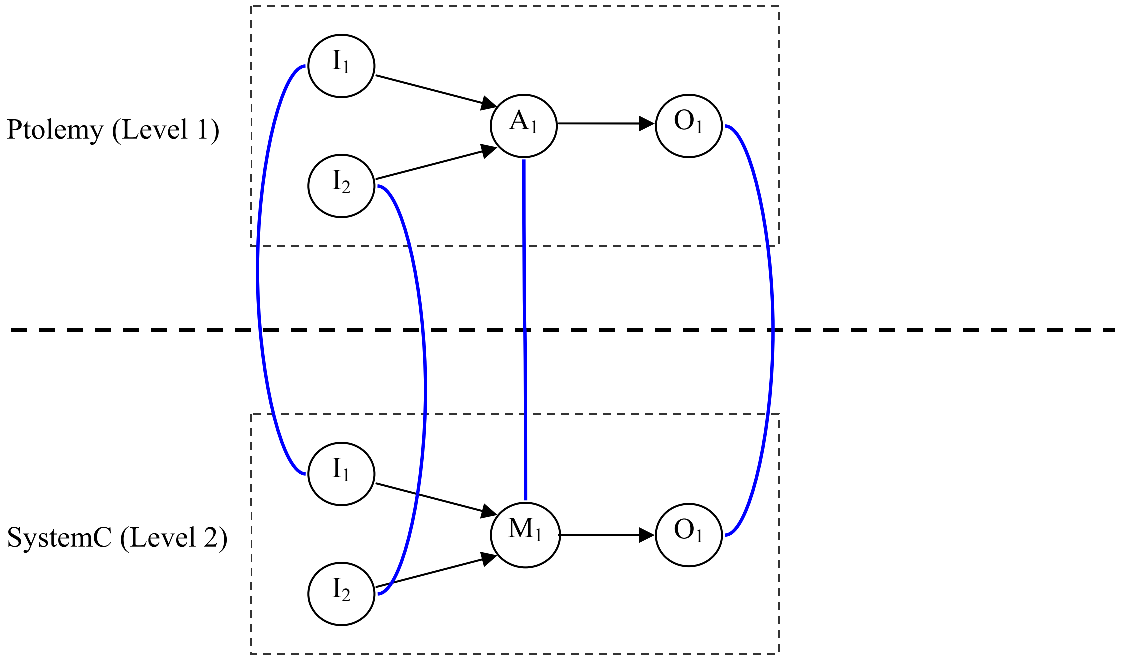


Figure 3.5. Bijection of nodes from Level 1 to Level 2

3.5 Mapping Level 2 to Levels 3 and 4

Given the definitions of digraphs of a SystemC model for both Level 3 and 4, we now define how to map a SystemC model of one domain onto a SystemC model of a different domain. The digraphs D_{L2} and D_{L3} represent two different SystemC domains. The mapping M_{2_34} of D_{L2} onto D_{L3} is as follows:

$$\begin{aligned} &M_{2\_34}: D_{L2} \Rightarrow D_{L3} \\ &F_{N2\_34}: N_{L2} \Rightarrow N_{L3} \qquad (3.5) \\ &F_{S2\_34}: C_{L2} \Rightarrow C_{L3} \end{aligned}$$

where F_{N2_34} and F_{S2_34} are defined as follows:

1) For F_{N2_34}, there was one case considered in this research.
   a. $N_{L2} < N_{L3}$. In our implementation, this mapping is one-to-one and not onto, or an injection [30]. Figure 3.6 illustrates this injective mapping.
2) For F_{S2_34}, the mapping functions of a channel *c* are as follows:
   a. F_{Sc2_34}: Performs an injection, or injective mapping, of all channels in D_{L2} to channels to D_{L3}.
   b. F_{Ssu2_34}: The storage/update mechanism for each channel in D_{L2} must be mapped to a channel storage/update mechanism in D_{L3}. Specifically, a SystemC model at Level 2 has a FIFO dataflow mechanism while a SystemC model at both Levels 3 and 4 has a signal dataflow mechanism.

3.6 Mapping Level 4 to Level 5

As seen in this chapter, there is a one-to-one mapping of Ptolemy internal actor (Level 1) to SystemC internal module (Level 2). This is the approach taken in this research. This is also the approach taken when going from Level 2 to Levels 3 and 4. In industry, it is common practice to perform one-to-many mapping in terms of component mapping when implementing either an ASIC or FPGA. This approach exploits design reuse to achieve the productivity necessary to build complex systems. Specifically, this is the case when moving down the final step in the 5-level design process, namely, Level 4 to Level 5. The one-to-many mapping is the result of the synthesis tool trying to represent the synthesis-ready Level 4 model through bottom-up design. The synthesis tool uses ASIC/FPGA target libraries to do so. The target library of choice contains standardized components (i.e.: adders, registers, logic gates, etc.) and it is these components the synthesis tool uses to represent the Level 5 model. Section 5.6 shows the result of a one-to-many mapping when moving from a Level 4 model to a Level 5 model.

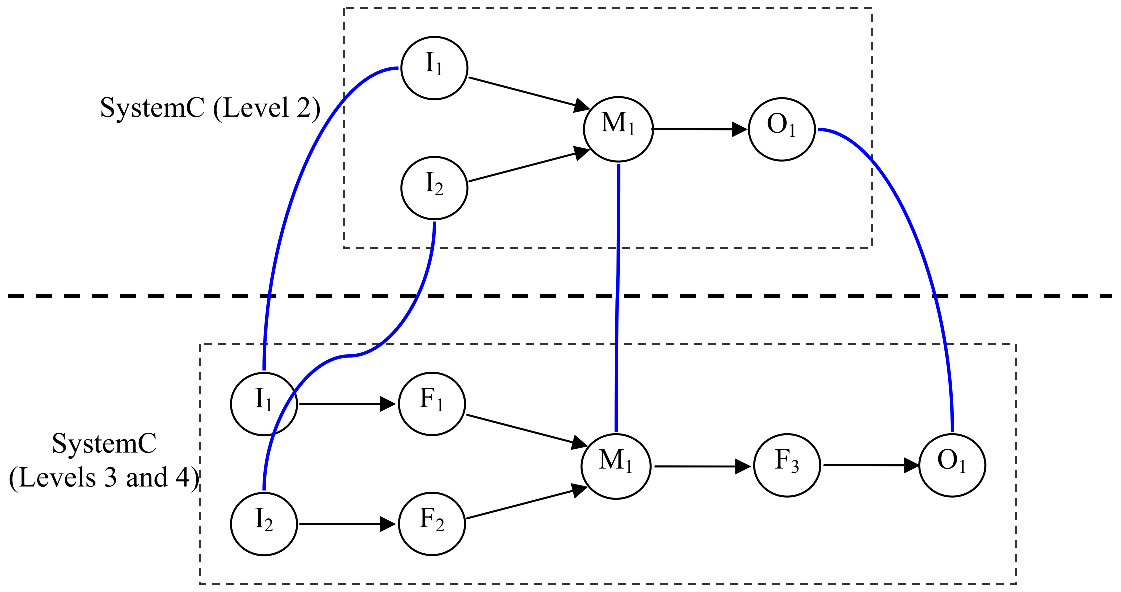


Figure 3.6. Injection of nodes from Level 2 to Levels 3 and 4

Chapter 4. XML Structures of Ptolemy Actors

To construct Ptolemy structural models, a graphical user interface (GUI) named *Vergil* is used [2]. These models are stored as XML documents. Specifically, *Vergil* stores models in ASCII files using an XML schema called MoML which stands for Modeling Markup Language. MoML is a modeling markup schema in the Extensible Markup Language (XML). It is intended for specifying interconnections of parameterized components [2]. These parameterized components are compiled Java code (e.g.: Java classes). Briefly stated, XML documents are used in Ptolemy to store the interconnection information of Ptolemy actors. Figure 4.1 shows a Ptolemy structural model containing three actors (e.g.: three parameterized components).

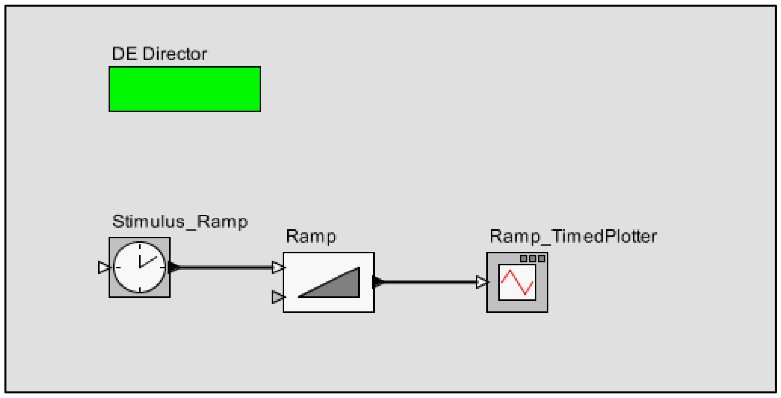


Figure 4.1. Ptolemy structural model constructed using *Vergil*

In this research, a methodology was defined to create XML documents that contain the Java code of Ptolemy actors. These XML documents are not to be confused with the XML documents of structural Ptolemy models described in the preceding paragraph. These actor-specific XML documents are to be used later on in the Ptolemy to SystemC translation. Section 4.1 briefly discusses the structure of a Ptolemy actor and how an actor performs its prescribed function. This section leads into section 4.2 which discusses the criteria for a Ptolemy actor to merit its own XML document and briefly discusses the case where an actor does not merit its own XML document. For that special case, a Perl script subroutine is used to achieve direct translation of the actor in the Ptolemy structural model to an equivalent SystemC module.

This chapter deals with describing the templates for actors. The sole purpose of using XML is to create SystemC module templates of Ptolemy actors. The SystemC module templates are a design library of templates to be used for the Ptolemy to SystemC translation.

4.1 Review of Functional Operations of Ptolemy Actors

To specify the operation of a Ptolemy actor, a set of public methods called *action methods* is used [2]. Ptolemy actors are composed of input/output (I/O) ports, action methods, parameters, and variables. Figure 4.2 shows a block diagram of an actor composition.

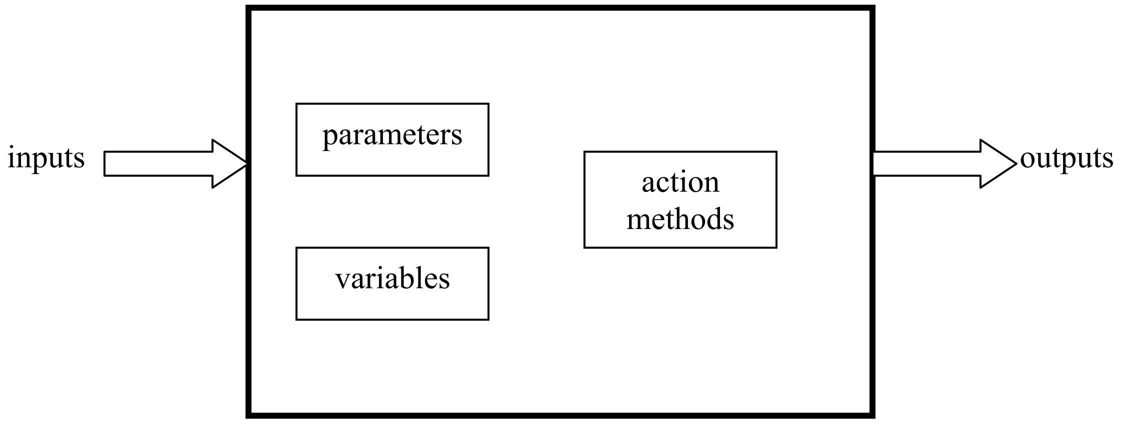


Figure 4.2. Overall composition of a Ptolemy actor

Generally, there are six action methods that functionally compose an actor. These six methods are *preinitialize()*, *initialize()*, *prefire()*, *fire()*, *postfire()*, and *wrapup()*, and they perform the functions as follows:

1) *preinitialize()*: set type constraints on variables;
2) *initialize()*: initialize the model (e.g. the state) at the beginning of simulation;
3) *prefire()*: determine if the model should fire;
4) *fire()*: read model inputs and perform model function;
5) *postfire()*: update model outputs, state, and schedule future firings (timed domains);
6) *wrapup()*: display final results.

In Ptolemy, the order in which the six action methods are executed is important. The *preinitialize()* and *initialize()* methods are first executed (once), in that order, while *prefire()*, *fire()*, and *postfire()* are executed repeatedly in that order. The *wrapup()* method displays final results [2].

4.2 Criteria for Constructing XML Documents of Ptolemy Actors

4.2.1 Review of SystemC Modules

In SystemC, the important parts of a module consist of a process and a constructor [4,9]. The process is the part of the SystemC code structure that defines the module function. The constructor performs the tasks as follows:

1) Declares the process type (i.e.: SC_METHOD, SC_THREAD, or SC_CTHREAD);
2) Declares triggering conditions needed to execute the process;
3) Initializes internal data structures (e.g.: variables) to known values.

To link the Java code structure of a Ptolemy actor to the C++ code structure of a SystemC module, the following transformations are required:

1) The contents of the Java *initialize()* method is placed in the SystemC constructor;
2) The contents of the *preinitialize()*, *prefire()*, *fire()*, and/or *postfire()* methods are placed in the body of the SystemC process.

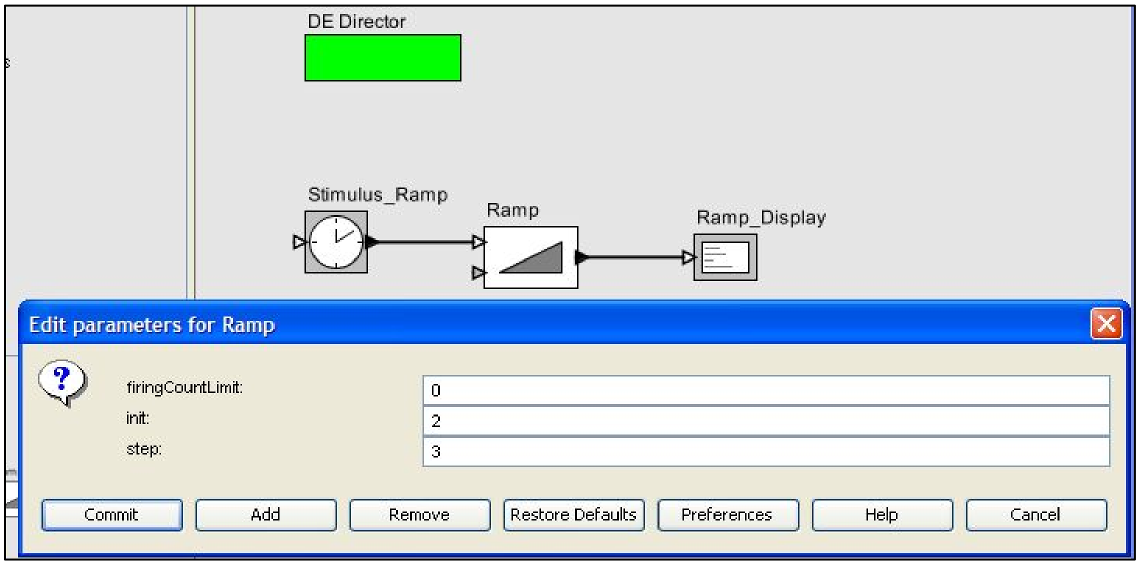
4.2.2 Requirements for Actor-specific XML Documents

For the Java code of an actor to merit its own XML document, three requirements must be met as follows:

1) The action methods that perform the actor's behavior must reside in the actor's Java class and not in any of its Java super classes [2,19,20]. Typically, the super class of an actor deals with GUI artifacts (i.e.: simulation control) and has no hardware counterpart. An example of an actor whose super classes deal with simulation control is the *Ramp* actor [2];
2) If an actor contains configurable parameters in *Vergil*, it must be value-configurable and not function-configurable. A value-configurable parameter is defined as a parameter that requires either an integer or Boolean data type value. Function-configurable is defined as a parameter that dictates a specific function that an actor is to perform. An example of an actor that is value-configurable is the *Ramp* actor [2]. Figure 4.3 shows the parameter window for this actor. The parameter names are firingCountLimit, init, and step. This actor is composed of three action methods (*initialize()*, *fire()*, and *postfire()*) which will be completely shown in section 4.3.2. An example of an actor that is function-

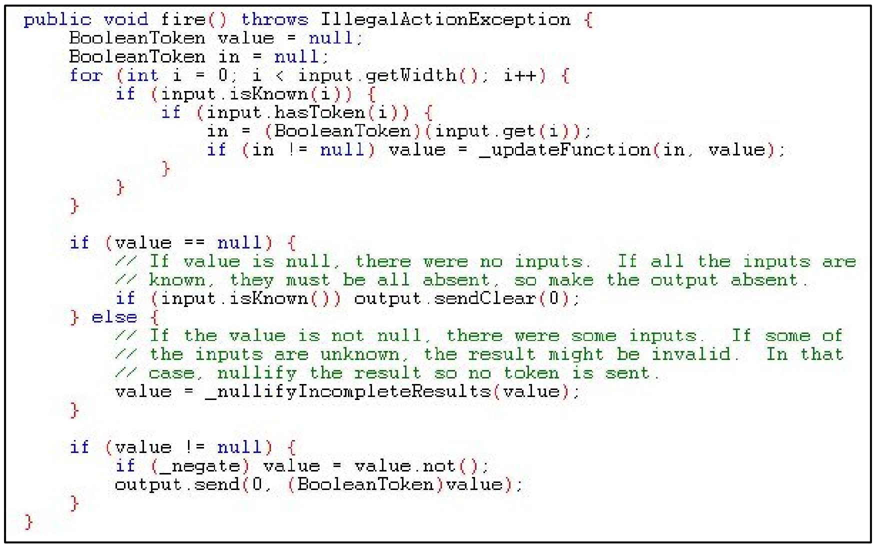
configurable and, hence, does not merit its own actor-specific XML document is the *NonStrictLogicFunction* actor [2]. This actor is basically composed of a *fire()* method and a function called *_updateFunction* found in a super class. The actor has a parameter that dictates which function out of a set of functions it is to perform. Figures 4.4 and 4.5 show the action method and function respectively. Figure 4.6 shows a Ptolemy structural model of this particular actor along with its function-configurable parameter window. Since this is deemed a function-configurable actor, a Perl script direct translation is better suited than an actor-specific XML document because the direct translation parses through the XML document of the Ptolemy structural model and constructs a SystemC module based solely on the selected function.

3) An actor must contain action methods. If an actor does not have (an) action method(s) then a Perl direct translation is required. This is because, with the approach used in this research, if an actor does not have action methods in its Java class, an actor-specific XML document can not be created. An example of an actor that does not have any of the prescribed action methods in its Java class is the *ModalModel* actor [2]. This actor models a finite state machine (FSM) [4,24,25].



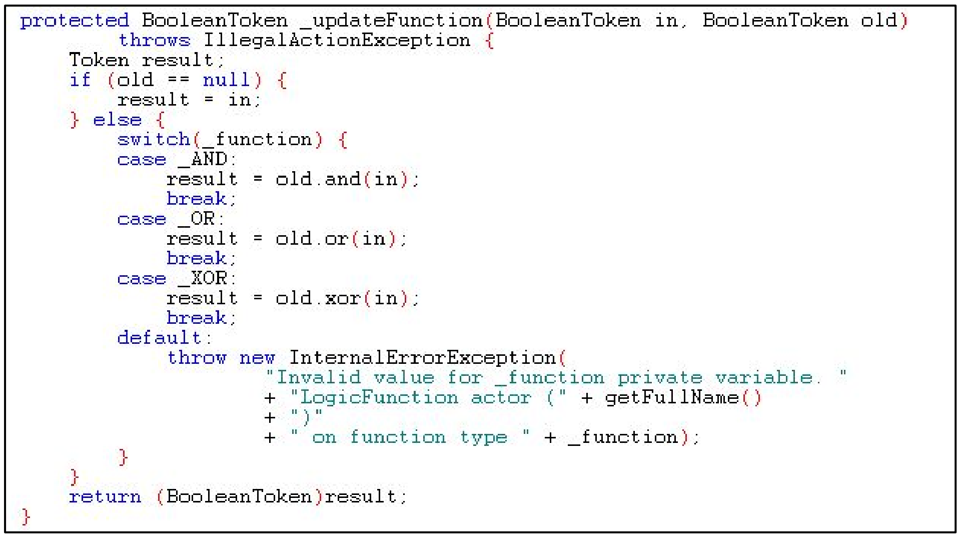
firingCountLimit: 0
init: 2
step: 3

Figure 4.3. Parameter window of *Ramp* actor for Ramp Test Model



```
public void fire() throws IllegalActionException {
    BooleanToken value = null;
    BooleanToken in = null;
    for (int i = 0; i < input.getWidth(); i++) {
        if (input.isKnown(i)) {
            if (input.hasToken(i)) {
                in = (BooleanToken)(input.get(i));
                if (in != null) value = _updateFunction(in, value);
            }
        }
    }

    if (value == null) {
        // If value is null, there were no inputs.  If all the inputs are
        // known, they must be all absent, so make the output absent.
        if (input.isKnown()) output.sendClear(0);
    } else {
        // If the value is not null, there were some inputs.  If some of
        // the inputs are unknown, the result might be invalid.  In that
        // case, nullify the result so no token is sent.
        value = _nullifyIncompleteResults(value);
    }

    if (value != null) {
        if (_negate) value = value.not();
        output.send(0, (BooleanToken)value);
    }
}
```

Figure 4.4. *fire()* action method of *NonStrictLogicFunction* actor



```
protected BooleanToken _updateFunction(BooleanToken in, BooleanToken old)
        throws IllegalActionException {
    Token result;
    if (old == null) {
        result = in;
    } else {
        switch(_function) {
        case _AND:
            result = old.and(in);
            break;
        case _OR:
            result = old.or(in);
            break;
        case _XOR:
            result = old.xor(in);
            break;
        default:
            throw new InternalErrorException(
                    "Invalid value for _function private variable. "
                    + "LogicFunction actor (" + getFullName()
                    + ")"
                    + " on function type " + _function);
        }
    }
    return (BooleanToken)result;
}
```

Figure 4.5. *_updateFunction* function in Java super class of *NonStrictLogicFunction* actor

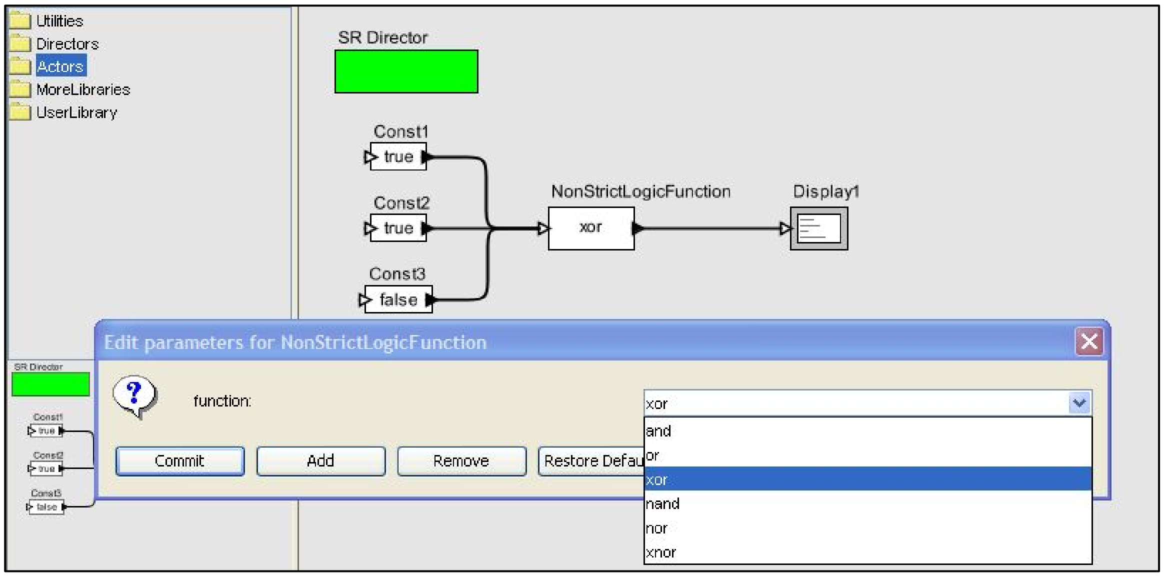


Figure 4.6. Ptolemy structural model for NonStrictLogicFunction Test Model

4.3 XML Representation of Ptolemy Actors

XML (Extensible Markup Language) is a language used to represent parse trees with each leaf containing data. XML is well-suited for formatting document types and for defining their structural relationships [15,16]. The ability to define tags and to create the proper relationships between information allows for storing and selective retrieving, or *parsing*, of data. Ptolemy actors are written in Java and follow a strict formatting style. The same applies to SystemC modules. Placing the Java code of a Ptolemy actor into an XML document facilitates conversion to a SystemC module.

XML documents follow rules specified by XML schemas. XML schemas describe and constrain the content of an XML document [15]. An XML schema was designed to represent the general Ptolemy code structure for Ptolemy actors. Specifically, in this research, it defines that the leaf elements contain the Java code of the six action methods whose code comes from either the class or super classes of the actor [19,20]. In addition, the leaf elements declare parameters and variables of the actor's class and/or super class(es).

4.3.1 XML Schema of Ptolemy Actors

Figure 4.7a illustrates the schema for a Ptolemy actor. Figure 4.7b illustrates the expanded *super_class* leaf element shown in Figure 4.7a. The elements *initialize*, *prefire*, *fire*, *postfire*, and *wrapup* in Figure 4.7a each contain three leaf elements named *local_top*, *local*, and *local_bottom*. This is illustrated in Figure 4.7c for the *initialize* element. The *preinitialize* element is not shown in Figures 4.7a and 4.7b for the sake of the reader to better visualize the tree structure of the schema with only five elements. However, the complete schema does contain the *preinitialize* element. The elements *local_top*, *local*, and *local_bottom* are needed for indicating where code from an action method of a super class is executed in the action method of a particular Ptolemy actor. For example, the use of *local_top* and *local_bottom* indicates that the action method code from the actor-specific super class is inherited after and before, respectively, the action method code contained in the actor being defined. The *local* element indicates that there is no super class to inherit and only the code of that particular method is invoked and executed [2,19,20].

For some Java coded actors, the I/O port names of a Ptolemy actor are defined in a super class (i.e.: *Ramp* actor). The schema in this research does not define the I/O names of an actor in the *super_class* element shown in Figure 4.7b. The I/O names are defined in the top-level schema shown in Figure 4.7a.

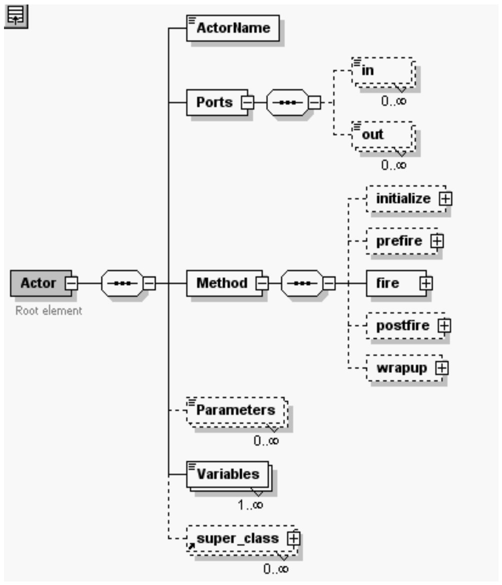


Figure 4.7a. XML Schema used for Ptolemy actors

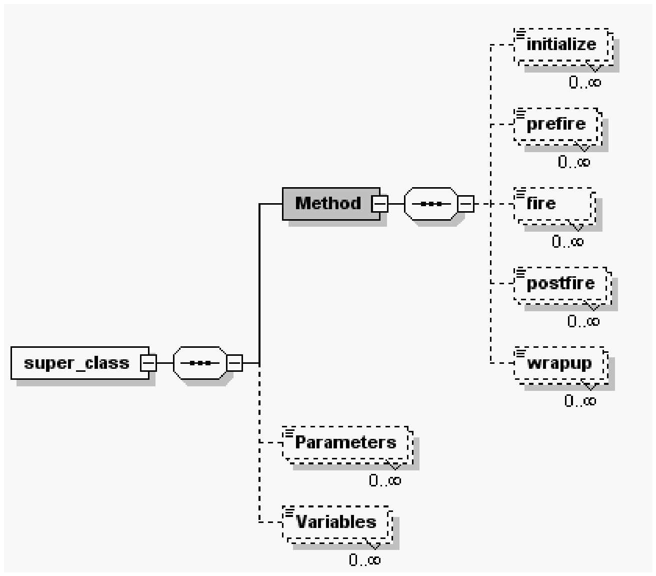


Figure 4.7b. XML Schema of *super_class* element

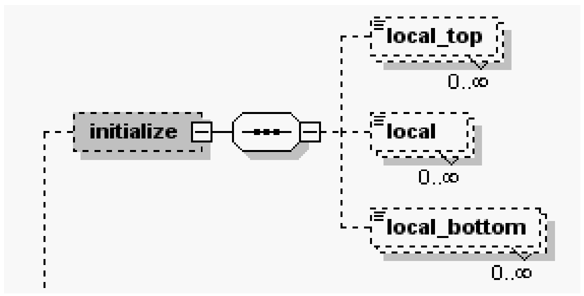
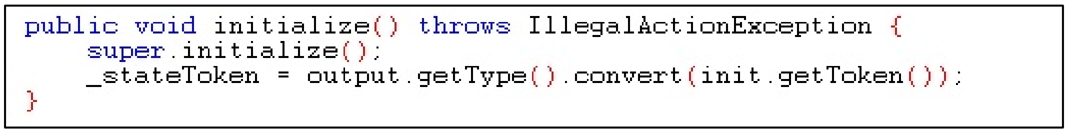


Figure 4.7c. XML view of *initialize* element in XML Schema

4.3.2 XML Document of Ptolemy Actors

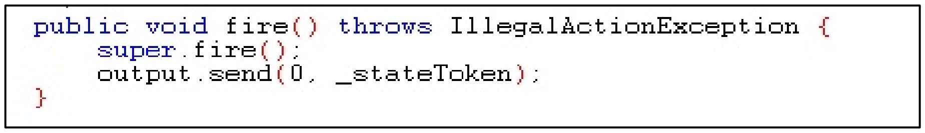
The schemas outlined in Figures 4.7a through 4.7c dictate how Ptolemy actor-specific XML documents are constructed. For example, the Ptolemy *Ramp* actor (see Figure 4.1) merits its own XML document based on the requirements in section 4.2.2. This actor consists of 2 input ports (*trigger* and *step*), 1 output port (*output*), 3 action methods (*initialize()*, *fire()*, and *postfire()*), and 2 super classes (*SequenceSource* and *Source*). Figures 4.8 through 4.10 show the Java code of each action method associated with the *Ramp* actor.

Figure 4.11 shows the XML document of the Ptolemy *Ramp* actor. Using the "designer rationale and technical common sense" paradigm stated in section 2.2.2, portions of each action method go into the actual XML document as can be seen in the *initialize*, *fire*, and *postfire* element tags. For this particular actor, the super classes will not be carried down the design process solely because those constructs represent GUI information specific to Ptolemy. But, in spite of that, the XML document of this actor is complete in that it conforms to the XML schema discussed and illustrated earlier in this chapter. It is these portions that are hardware-realizable and are carried down the design process.



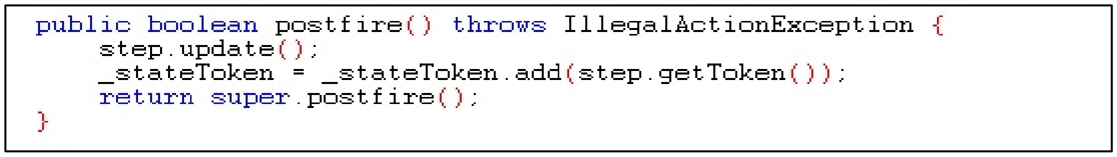
```
public void initialize() throws IllegalActionException {
    super.initialize();
    _stateToken = output.getType().convert(init.getToken());
}
```

Figure 4.8. *initialize()* action method of *Ramp* actor



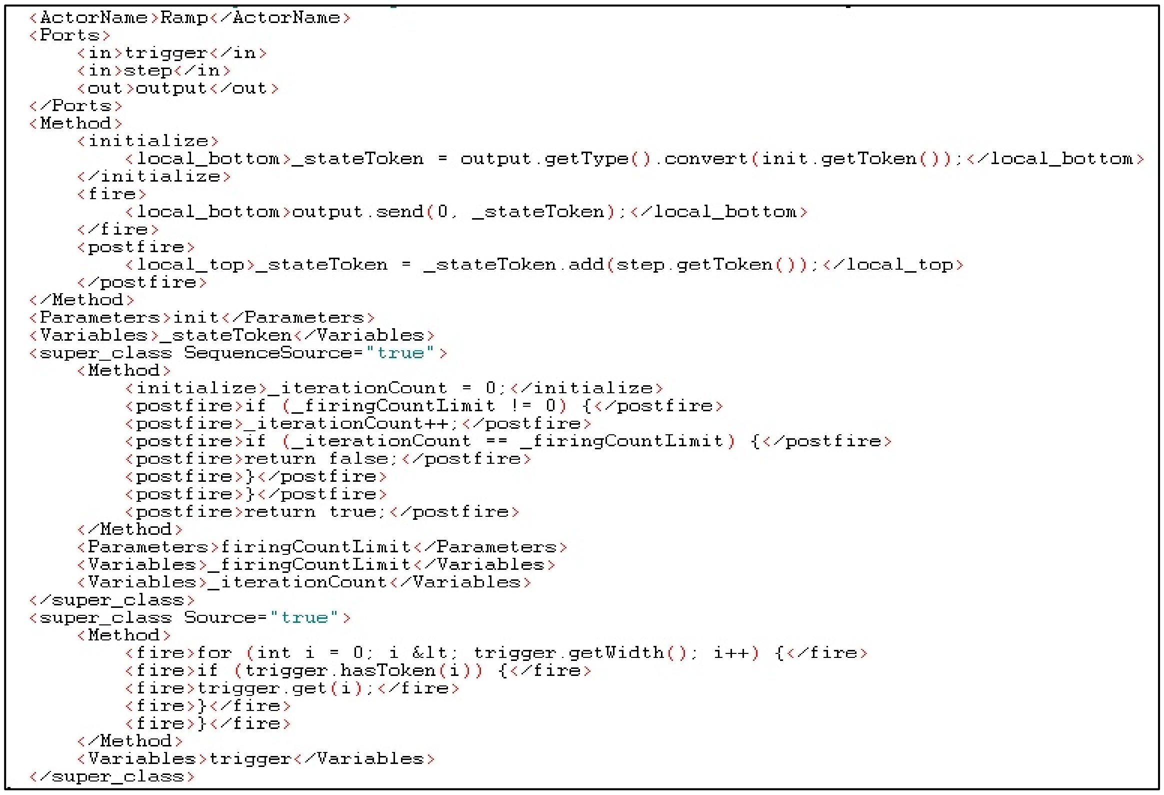
```
public void fire() throws IllegalActionException {
    super.fire();
    output.send(0, _stateToken);
}
```

Figure 4.9. *fire()* action method of *Ramp* actor



```
public boolean postfire() throws IllegalActionException {
    step.update();
    _stateToken = _stateToken.add(step.getToken());
    return super.postfire();
}
```

Figure 4.10. *postfire()* action method of *Ramp* actor



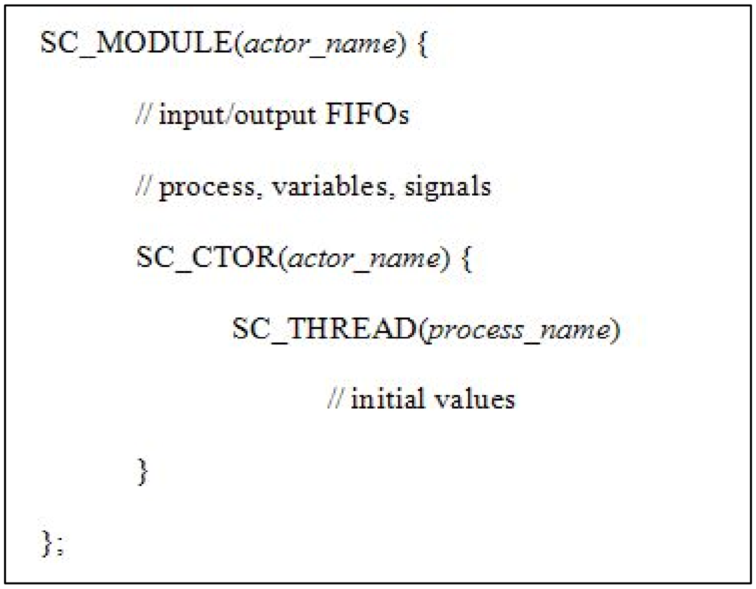
```
<ActorName>Ramp</ActorName>
<Ports>
    <in>trigger</in>
    <in>step</in>
    <out>output</out>
</Ports>
<Method>
    <initialize>
        <local_bottom>_stateToken = output.getType().convert(init.getToken());</local_bottom>
    </initialize>
    <fire>
        <local_bottom>output.send(0, _stateToken);</local_bottom>
    </fire>
    <postfire>
        <local_top>_stateToken = _stateToken.add(step.getToken());</local_top>
    </postfire>
</Method>
<Parameters>init</Parameters>
<Variables>_stateToken</Variables>
<super_class SequenceSource="true">
    <Method>
        <initialize>_iterationCount = 0;</initialize>
        <postfire>if (_firingCountLimit != 0) {</postfire>
        <postfire>_iterationCount++;</postfire>
        <postfire>if (_iterationCount == _firingCountLimit) {</postfire>
        <postfire>return false;</postfire>
        <postfire>}</postfire>
        <postfire>}</postfire>
        <postfire>return true;</postfire>
    </Method>
    <Parameters>firingCountLimit</Parameters>
    <Variables>_firingCountLimit</Variables>
    <Variables>_iterationCount</Variables>
</super_class>
<super_class Source="true">
    <Method>
        <fire>for (int i = 0; i &lt; trigger.getWidth(); i++) {</fire>
        <fire>if (trigger.hasToken(i)) {</fire>
        <fire>trigger.get(i);</fire>
        <fire>}</fire>
        <fire>}</fire>
    </Method>
    <Variables>trigger</Variables>
</super_class>
```

Figure 4.11. XML document of Ptolemy *Ramp* actor

4.4 SystemC Module Template of XML Document of Ptolemy Actors

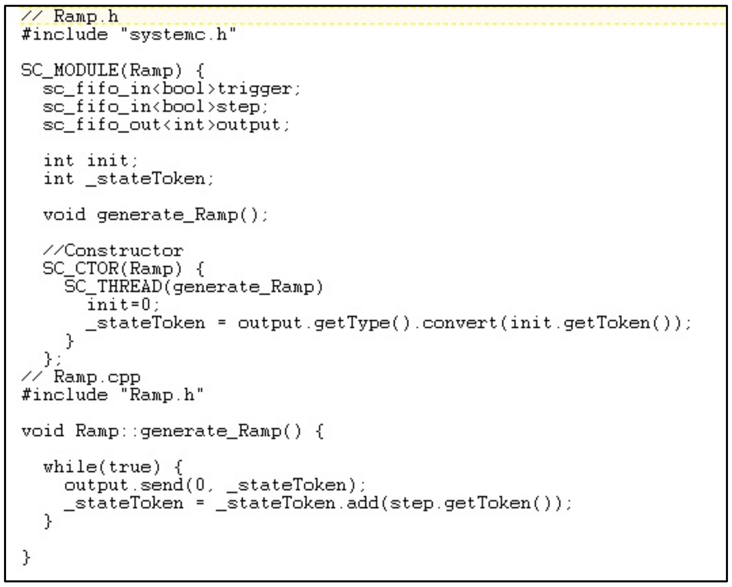
Now that the XML document of a Ptolemy actor is created, the next step is to produce a SystemC module template. As stated earlier in this chapter, these templates will serve as a design library of templates. To create these templates, an XML tool called XSLT (the Extensible Stylesheet Language Transformations) is used. Since XML documents are basically parse trees, XSLT stylesheets are used to manipulate and transform XML structures. These stylesheets extract and transform XML documents into any format (i.e.: HTML document, text file, another XML document, etc.) [15-17].

For this research, an XSLT stylesheet is used to parse through an XML document of an actor and create a SystemC module template in the general format shown in Figure 4.12. In keeping with the considerations stated in section 2.2.2 and requirements in section 4.2.2, the XSLT style sheet ignores the super class tags in the XML document of the Ptolemy *Ramp* actor because they are the actor's Ptolemy GUI references. As stated in Chapter 2 for Level 2 representation in the 5-level design process, SystemC modules consist of *sc_fifo* and SC_THREAD constructs. The stylesheet parses through an XML document and translates it into a SystemC module template with those corresponding channel and process type. The result of this parsing and translating is shown in Figure 4.13 where the XSLT stylesheet created in this research transforms the XML document shown in Figure 4.11.



```
SC_MODULE(actor_name) {

        // input/output FIFOs

        // process, variables, signals

        SC_CTOR(actor_name) {

                SC_THREAD(process_name)

                        // initial values

        }

};
```

Figure 4.12. General format for SystemC module template



```
// Ramp.h
#include "systemc.h"

SC_MODULE(Ramp) {
  sc_fifo_in<bool>trigger;
  sc_fifo_in<bool>step;
  sc_fifo_out<int>output;

  int init;
  int _stateToken;

  void generate_Ramp();

  //Constructor
  SC_CTOR(Ramp) {
    SC_THREAD(generate_Ramp)
      init=0;
      _stateToken = output.getType().convert(init.getToken());
    }
  };
// Ramp.cpp
#include "Ramp.h"

void Ramp::generate_Ramp() {

  while(true) {
    output.send(0, _stateToken);
    _stateToken = _stateToken.add(step.getToken());
  }

}
```

Figure 4.13. SystemC module template of XML document of Ptolemy *Ramp* actor

The XSLT stylesheet created in this research produces SystemC module templates that serve as a design library of templates. Although the stylesheet serves this crucial step, needed for further

Ptolemy to SystemC translation, the stylesheet does not manipulate the actual Java code content associated in the element tags of XML documents. In addition, if the element tag in an XML document contains a default value of a parameter, the stylesheet does nothing to it but translate it as is to the corresponding SystemC module template. In other words, Java syntax/semantics and default parameter values are preserved when the stylesheet creates a template from the XML document of a Ptolemy actor. To translate the Java syntax/semantics of a Ptolemy actor to C++ syntax/semantics of a SystemC module and replace default parameter values with actual parameter values in a Ptolemy model, we chose the text manipulation programming language Perl.

Chapter 5. Transformation Methodology for Ptolemy Models to SystemC Models

Since transforming Ptolemy models to SystemC modules entails manipulating SystemC module templates and XML documents of Ptolemy structural models, the text manipulation program Perl is the crucial implementation tool used for the desired Ptolemy to SystemC translation. Specifically, Perl takes these templates and XML documents as input and translates them into a SystemC model consisting of source and header files. Perl is chosen because of its inherent ease of string manipulation [21,22].

To fulfill the transformation methodology, three Perl scripts, are created. The first script takes a Ptolemy structural model as input and produces an initial SystemC model by performing the two primary functions as follows:

1) Convert the Java syntax/semantics of a Ptolemy model to the C++ syntax/semantics of a SystemC model;
2) If necessary, replace default parameter values of SystemC module templates with specific parameter values associated with Ptolemy actors in a Ptolemy structural model.

In other words, this script goes from Level 1 to Level 2 in the design process. The second script takes the Level 2 SystemC initial representation and replaces the non-synthesizable SystemC constructs with synthesizable constructs. This yields a Level 3 model which represents an unconstrained, synthesis-ready SystemC model. To implement the hardware considerations stated in section 2.2.1, the third and final Perl script takes this Level 3 model and produces a constrained, synthesis-ready SystemC model. The result is a Level 4 model.

In this chapter, the manner in which Ptolemy actors are characterized for correct transformation to Level 2 and Level 3 is discussed. As stated in section 2.1, Level 2 models are SystemC modules that use threaded processes (SC_THREAD), while Level 3 models are SystemC models that operate with the method process (SC_METHOD). This is followed by discussing the general outline structure of a complete SystemC model associated with RTL modeling. Lastly, this chapter discusses the step-by-step methodology (using Perl scripts) to translate a Ptolemy structural model into a Level 4 model which is a synthesis-ready SystemC model, the final goal of the translation process.

5.1 Characterization of Ptolemy Actors for Translation

When translating a Level 1 (Ptolemy) model to a Level 2 model, characterizing the Ptolemy model is important to ensure that the semantics of a Ptolemy actor matches the semantics of a SystemC module. Since a Level 2 model preserves the token dataflow mechanism used in a Ptolemy structural model, this research characterizes the Ptolemy model and its target SystemC model as being in the *delayless* or *with delay* category. A Level 2 model that falls under the *delayless* category does not require the insertion of control logic when translating the appropriate module template. But, under the *with delay* category, the insertion of control logic is necessary. This is because, when proceeding down the 5-level design process, the effects of the Ptolemy director in terms of its execution and communication semantics must be preserved. Control of a Ptolemy actor is dictated outside the actor via the director while control of a SystemC module is dictated inside the module.

When translating a Level 2 model into a Level 3 model, emphasis is placed on realizing an RTL model. The characterization of the Level 2 model (e.g.: *delayless* or *with delay* category) is preserved during the transformation to Level 3. Along with preserving this characterization, there are two technical considerations when moving to this level and they are as follows:

1) Signal Flow Mechanism: The semantics of the signal dataflow mechanism associated with RTL modeling must implement the same behavior achieved by the token dataflow mechanism. This is a natural requirement as one moves closer to hardware. Namely, when one proceeds down the hierarchy, semantics are frequently changed.
2) Reset Mechanism: If a Level 2 model has sequential behavior [23,24], the concept of a reset mechanism associated with RTL modeling must be considered.

5.1.1 Delayless Category for Level 2 Models

For a Ptolemy actor, an input sequence produces an output sequence. If the same input sequence is repeated and yields the same output sequence, regardless of the nature of the sequence, the actor falls under the *delayless* category. This category is in reference to combinational logic where a system's present output values are strictly dependent on the system's present input values [23]. Examples of Ptolemy actors that fall under this category are the *AddSubtract* and *MultiplyDivide* actors [2]. Figure 5.1a shows a Ptolemy structural model of an *AddSubtract* actor configured to perform addition. As shown in the sink actors *Input_Display1* and *Input_Display2*, the source actors *Input_Sequence1* and *Input_Sequence2* repeat the input sequence *4,12,4,87* and *22,-7,55,9* respectively. The sink actor *Adder_Output* displays two identical output responses of the *AddSubtract* actor to the repeated input sequences. Figures 5.1b illustrates these results. For this category, control logic is not necessary for the initial module template to Level 2 model translation. The only translation that the Perl script performs is the first primary function outlined in the beginning of this chapter. The module template and Level 2 model of this actor is shown later in this chapter.

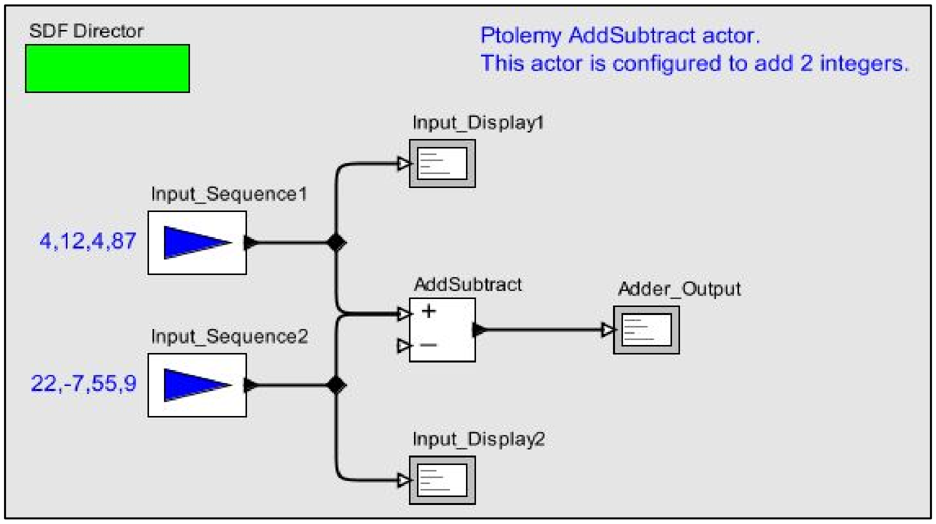


Figure 5.1a. *Vergil* window for AddSubtract Test Model

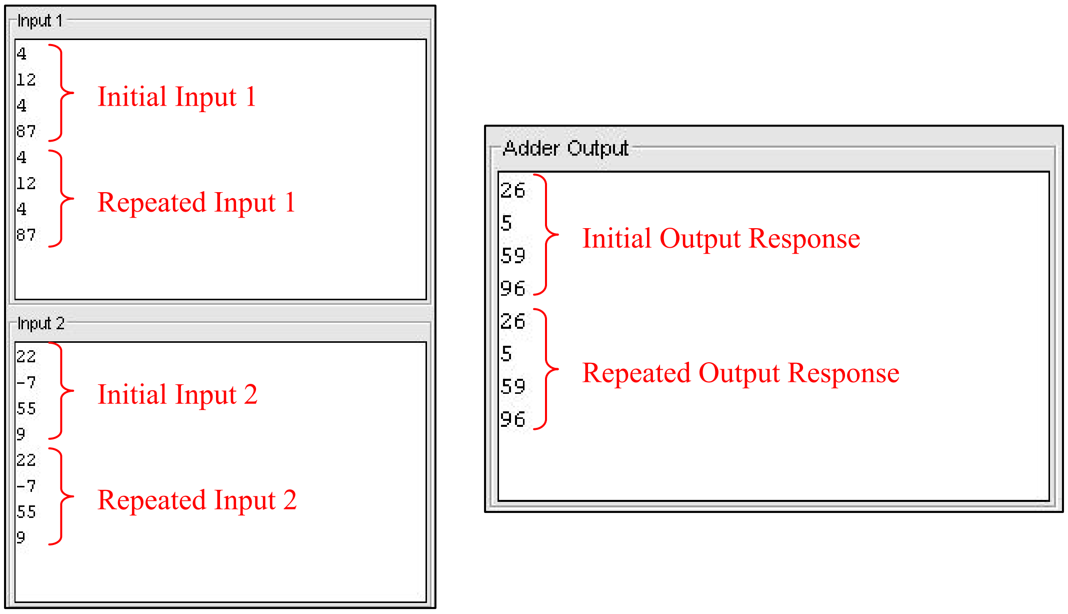

Input 1
```
4
12
4
87
4
12
4
87
```
Input 2
```
22
-7
55
9
22
-7
55
9
```
Adder Output
```
26
5
59
96
26
5
59
96
```

Figure 5.1b. I/O values of AddSubtract Test Model

5.1.2 With Delay Category for Level 2 Models

In keeping with the same repeated input sequence procedure stated in section 5.1.1, if the second output response of a Ptolemy actor is not identical to the initial output sequence, the actor falls under the *with delay* category. Examples of actors that fall under this category are the *SampleDelay* and *NonStrictDelay* actors [2]. Figure 5.2a shows the Ptolemy structural model of a *SampleDelay* actor configured to output two zeros as initial values before passing the input sequence *1,2,3,4,5*. Note, this actor does not necessarily have to initialize with a vector of zeros. The source and sink actors *Input_Sequence* and *Output_Display* produce the input sequence and display the output responses, respectively, as can be seen in Figure 5.2b. As shown in this figure, the two output responses are dissimilar. Regardless of the initial two input values, the *SampleDelay* actor outputs two consecutive zeros initially to start. In Ptolemy, this is also a means of specifying delay which. in this case, is a delay of two samples. The module template of this actor requires the Perl script to insert control logic when producing a Level 2 model because translating the model as is yields incorrect results. That is because the Ptolemy director provides the control needed for the *SampleDelay* actor to perform its correct behavior. The module template along with its translated SystemC module (Level 2 model) for this particular actor is shown later in this chapter.

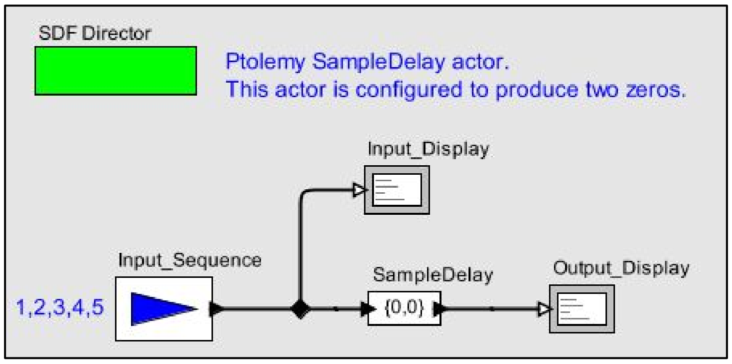


Figure 5.2a. *Vergil* window for SampleDelay Test Model

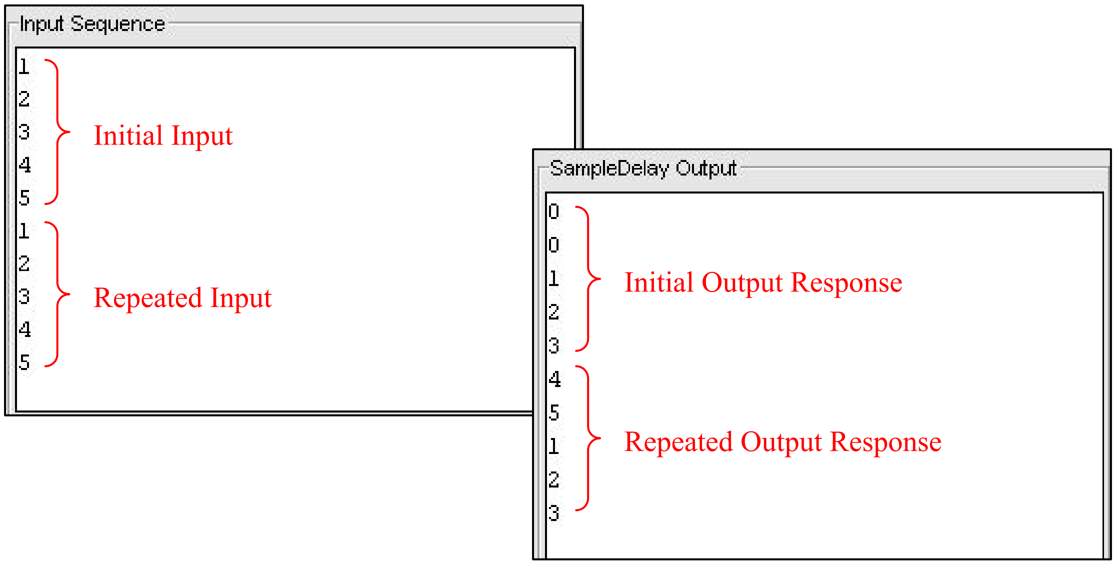


Figure 5.2b. I/O values of SampleDelay Test Model

5.1.3 Behavior Preservation of Level 3 Models under RTL Modeling

The Level 3 model in the 5-level design process utilizes RTL modeling, and, because of that, the Signal Flow Mechanism consideration stated in the opening section of section 5.1 comes into play. In most cases in this research, simply replacing the *sc_fifo* constructs (Level 2) with *sc_signal* constructs (Level 3) is sufficient. It is sufficient provided that the replacement preserves functional behavior when going from a Level 2 model to a Level 3 model. But, for the situation where that same construct replacement step does not preserve behavior, this research chose a convention to ensure that level-to-level behavior preservation. Specifically, there are two cases. In one case, the aforementioned construct replacement step is sufficient, while, in the other case, it is not sufficient.

Case 1 (Not sufficient): There are certain Ptolemy actors that, because of the token dataflow mechanism prescribed to in Ptolemy, a direct translation as is from a Level 2 to Level 3 model is not sufficient. It is not sufficient because the communication mechanisms of Level 2 and Level 3 models are different. Level 2 models have a token dataflow mechanism while Level 3 models have a signal dataflow mechanism. For a Ptolemy actor, an input token will always be processed regardless whether the current input token is equal or not equal to the previous input token. For a Level 3 model, if the SystemC module is sensitive to an input signal, that signal will only be processed if there is a signal change [4,5,10]. This is the difference in semantics between token dataflow and signal dataflow. To ensure that a SystemC module would always process a signal regardless of whether it changes or not, this research chose the convention of incorporating a clock signal into the Level 3 model. This ensures that whenever the Level 3 model encounters a clock signal, the input signal will always be processed. In addition, this convention explicitly implies that the resulting model has sequential behavior. With this behavior, the Reset Mechanism consideration comes into play. To incorporate this mechanism into a Level 3 model, the two steps must be performed as follows:

If the header file portion of the SystemC module template has variables or output ports, reset those parameters to either a null value (i.e.: zero or Boolean false) or a non-zero initial condition in the source file of the SystemC module, depending on the actor and data type.

An example of an actor that requires handling of the Signal Flow Mechanism and Reset Mechanism considerations is the *Ramp* actor [2]. This actor is enabled whenever a Boolean token is present on the trigger input regardless of whether a token is true or false. Thus, a continuous stream of tokens keeps the *Ramp* actor enabled and it increments/decrements when each token is received. In this research, the RTL equivalent of this actor is a clocked counter with a Boolean enable input [23,24]. Whenever the enable input is true, the counter (Ramp model) decrements/increments on the rising edge of the clock. Figure 5.3a shows a Ptolemy structural model of a *Ramp* actor receiving Boolean true tokens. In this model, this actor is configured to initially start at integer 2 and increment up by one whenever a true token is received. The source actor *Token_Source* produces the stream of true tokens while the sink actors *Token_Source_Display* and *Ramp_Display* display the stream of true tokens and ramp output of the *Token_Source* and *Ramp* actors, respectively. Figure 5.3b shows the plots of these two sink actors. As can be seen by the plots, with every true token, a ramp output increments.

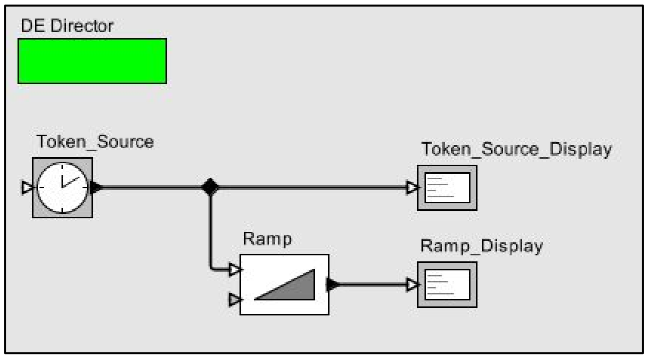
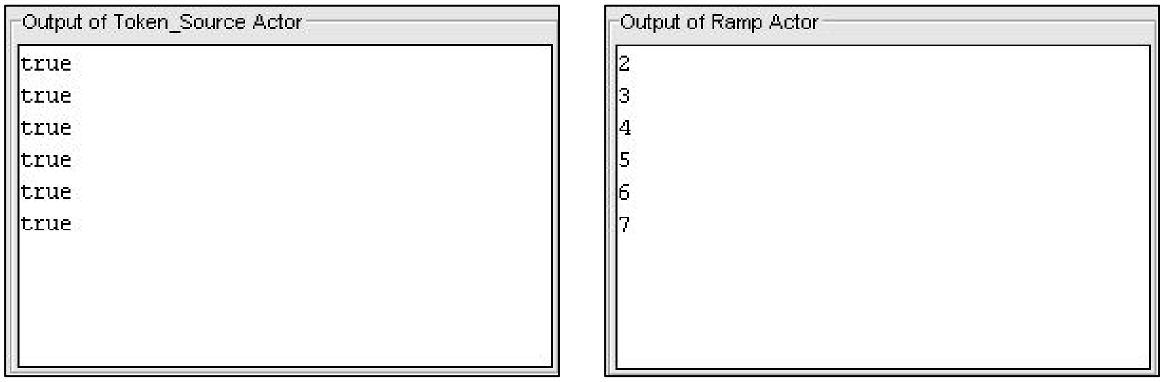


Figure 5.3a. *Vergil* window for Ramp Test Model



```
true
true
true
true
true
true
```

```
2
3
4
5
6
7
```

Figure 5.3b. Plots of *Token_Source* and *Ramp* actors

Figure 5.4 shows the RTL waveform of the RTL version of Figure 5.3a. In this research, the chosen characteristics of RTL modeling for Level 3 models are as follows:

1) For the Reset Mechanism consideration, reset is asynchronous and active high;
2) Sequential models use clocks that are rising edge triggered;
3) I/O registering are implemented using D-type flip-flops.

As can be seen in Figure 5.4, the signal *SystemC.input* and *SystemC.output[31:0]* are the input and output ports, respectively. The reset port is *SystemC.reset*. The RTL waveform is explained as follows:

1) *SystemC.input_reg* and *SystemC.output[31:0]* are the outputs of the I/O D-type flip-flops. Whenever *SystemC.reset* is high, these outputs are set to zero.
2) *SystemC.output_reg[31:0]* is the output of the Ramp model (Level 3). As stated earlier, the Ramp model has a clock port and reset port. Therefore, *SystemC.output_reg[31:0]* is also set to zero when reset is high.

3) When *SystemC.reset* is low and *SystemC.input* is high, *SystemC.input* is clocked in at the next rising edge of the clock *SystemC.clock_tb* on to *SystemC.input_reg*. *SystemC.input_reg* is the enable input to the Ramp model.
4) *SystemC.output_reg[31:0]* initially outputs the integer 2 and increments up on every rising edge of the clock. This happens when the enable input, *SystemC.input_reg*, is high.

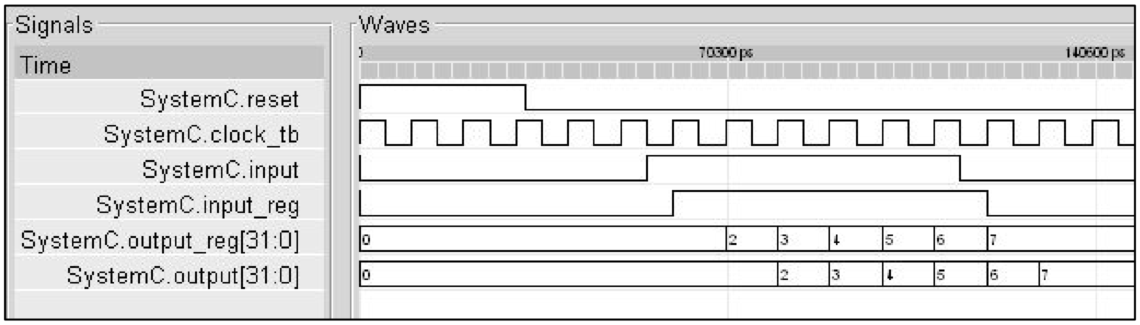
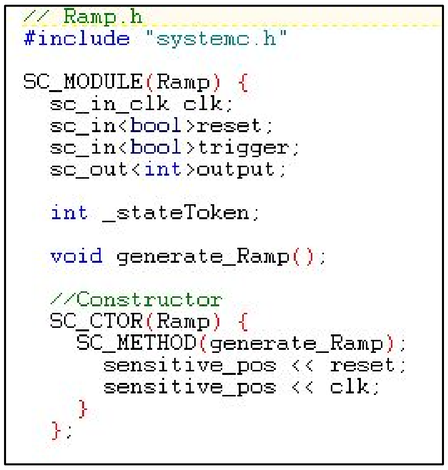
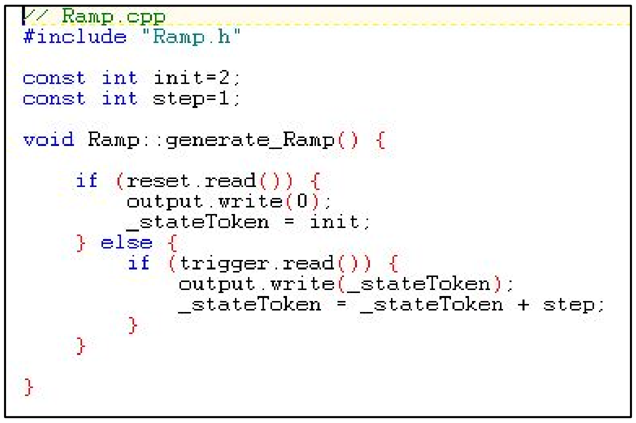


Figure 5.4. RTL waveform of Ramp model

The Ramp model illustrates the special case where simply replacing *sc_fifo* constructs with *sc_signal* constructs does not preserve behavior. The SystemC module of the Ramp model is shown in Figure 5.5.



```
// Ramp.h
#include "systemc.h"

SC_MODULE(Ramp) {
  sc_in_clk clk;
  sc_in<bool>reset;
  sc_in<bool>trigger;
  sc_out<int>output;

  int _stateToken;

  void generate_Ramp();

  //Constructor
  SC_CTOR(Ramp) {
    SC_METHOD(generate_Ramp);
      sensitive_pos << reset;
      sensitive_pos << clk;
    }
  };
```

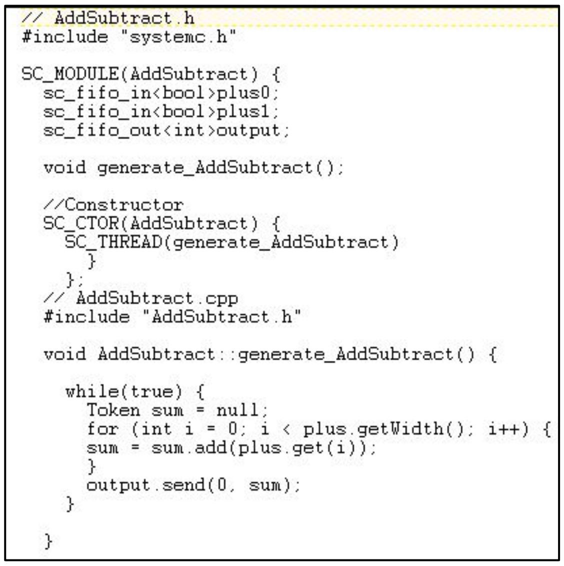


```
// Ramp.cpp
#include "Ramp.h"

const int init=2;
const int step=1;

void Ramp::generate_Ramp() {

    if (reset.read()) {
        output.write(0);
        _stateToken = init;
    } else {
        if (trigger.read()) {
            output.write(_stateToken);
            _stateToken = _stateToken + step;
        }
    }

}
```

Figure 5.5. SystemC header and source files for Ramp module (Level 3)

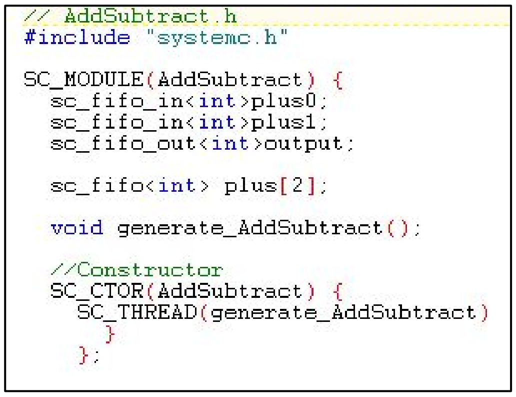
Case 2 (Sufficient): An example of a model where the construct replacement step is sufficient is the *AddSubtract* actor shown in Figure 5.1a. The module template and Levels 2 and 3 models of this actor are shown in Figures 5.6 through 5.8, respectively. This actor does not require the Reset

Mechanism consideration because of the aforementioned sufficiency of the construct replacement step.

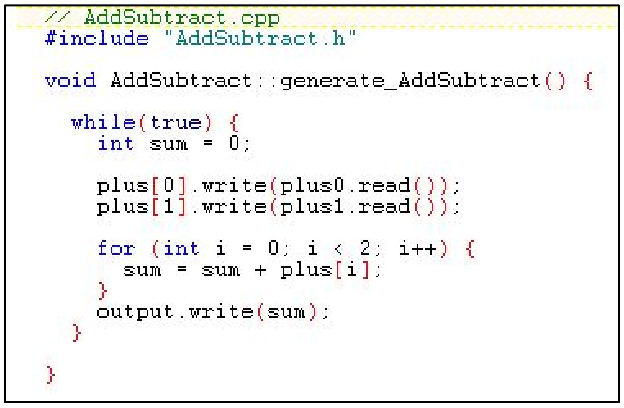


```
// AddSubtract.h
#include "systemc.h"

SC_MODULE(AddSubtract) {
  sc_fifo_in<bool>plus0;
  sc_fifo_in<bool>plus1;
  sc_fifo_out<int>output;

  void generate_AddSubtract();

  //Constructor
  SC_CTOR(AddSubtract) {
    SC_THREAD(generate_AddSubtract)
      }
    };
  // AddSubtract.cpp
  #include "AddSubtract.h"

  void AddSubtract::generate_AddSubtract() {

    while(true) {
      Token sum = null;
      for (int i = 0; i < plus.getWidth(); i++) {
      sum = sum.add(plus.get(i));
      }
      output.send(0, sum);
    }

  }
```

Figure 5.6. SystemC module template of Ptolemy *AddSubtract* actor

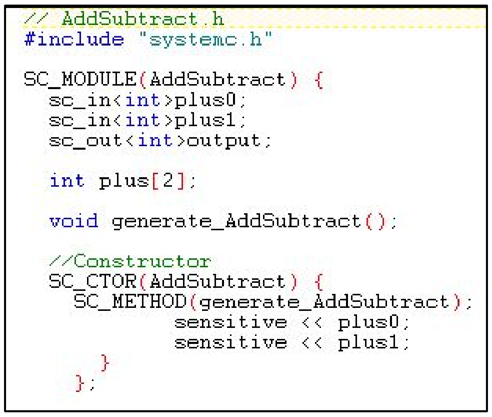


```
// AddSubtract.h
#include "systemc.h"

SC_MODULE(AddSubtract) {
  sc_fifo_in<int>plus0;
  sc_fifo_in<int>plus1;
  sc_fifo_out<int>output;

  sc_fifo<int> plus[2];

  void generate_AddSubtract();

  //Constructor
  SC_CTOR(AddSubtract) {
    SC_THREAD(generate_AddSubtract)
      }
    };
```

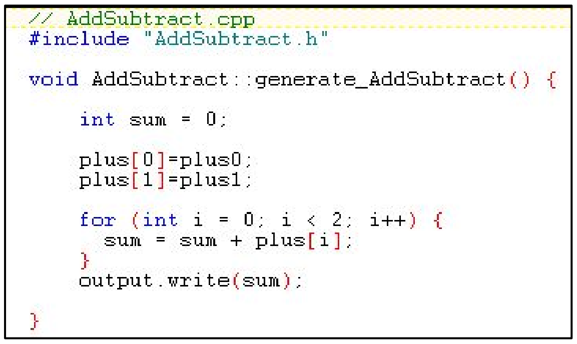


```
// AddSubtract.cpp
#include "AddSubtract.h"

void AddSubtract::generate_AddSubtract() {

  while(true) {
    int sum = 0;

    plus[0].write(plus0.read());
    plus[1].write(plus1.read());

    for (int i = 0; i < 2; i++) {
      sum = sum + plus[i];
    }
    output.write(sum);
  }

}
```

Figure 5.7. SystemC module for Level 2 model of Ptolemy *AddSubtract* actor



```
// AddSubtract.h
#include "systemc.h"

SC_MODULE(AddSubtract) {
  sc_in<int>plus0;
  sc_in<int>plus1;
  sc_out<int>output;

  int plus[2];

  void generate_AddSubtract();

  //Constructor
  SC_CTOR(AddSubtract) {
    SC_METHOD(generate_AddSubtract);
            sensitive << plus0;
            sensitive << plus1;
    }
  };
```



```
// AddSubtract.cpp
#include "AddSubtract.h"

void AddSubtract::generate_AddSubtract() {

    int sum = 0;

    plus[0]=plus0;
    plus[1]=plus1;

    for (int i = 0; i < 2; i++) {
      sum = sum + plus[i];
    }
    output.write(sum);

}
```

Figure 5.8. SystemC module for Level 3 model of Ptolemy *AddSubtract* actor

5.2 RTL Structure of a Complete SystemC Model

Because of the RTL modeling style, a complete SystemC model generally follows the mathematical structure shown in Figure 3.3. Specifically, in the graphical sense, this complete structure does not include source modules, sink modules, and their respective incident channels. This concept is better viewed by taking a Ptolemy structural model and performing the tasks as follows:

1) Replace all source and sink actors with SystemC D-type flip-flops (register banks);
2) Replace Ptolemy actors with corresponding SystemC modules;
3) Replace all Ptolemy relations with SystemC *sc_signal* channels.

Figure 5.9 illustrates the RTL version of a Ptolemy model that follows the above three tasks. Specifically, this is the RTL structure of the digraph shown in Figure 3.3. As stated in section 5.1.3, D-type flip-flops are rising-edge triggered and each flip-flop has an asynchronous, active-high reset port.

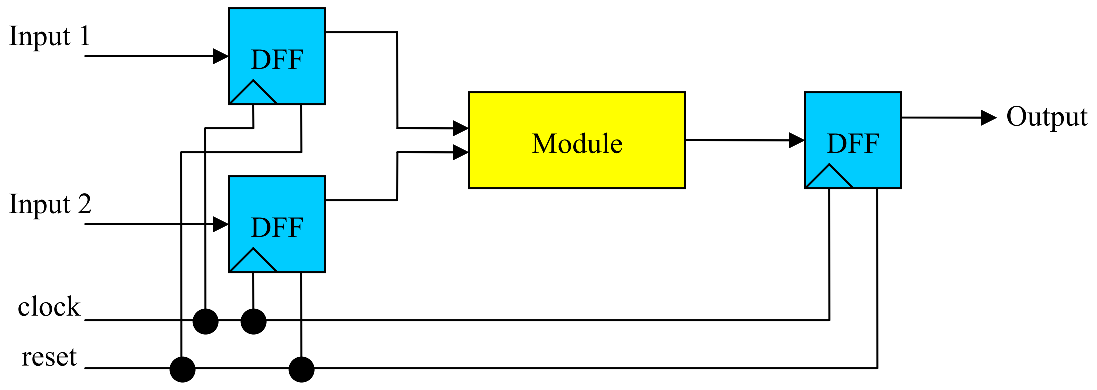


Figure 5.9. RTL version of a Ptolemy structural model

The three Perl scripts mentioned in the introduction section of this chapter accomplishes the goal of translating a Ptolemy model into the above RTL model shown in Figure 5.9. The remaining sections in this chapter show how the Ptolemy to SystemC transformation is achieved as each of the scripts proceeds down the 5-level design process outlined in Chapter 2. Each level in the design process is functionally verified through simulation. The simulation results of the Ptolemy structural model (Level 1) is the benchmark against which the simulation results of Levels 2 through 4 are compared.

5.3 Level 1 to Level 2 Translation

The first Perl script performs the Level 1 to Level 2 translation. To do so, three tasks are executed. These three tasks specifically handle the two primary functions outlined in the introductory section of this chapter. As stated in section 2.1.1, moving to Level 2 introduces the Kahn process network MOC for the initial SystemC model.

5.3.1 Task 1: Parsing of XML document of a Ptolemy Structural Model

In this initial task, the Perl script performs three operations. The operations are as follows:

1) Capture director name (MOC);
2) Capture Ptolemy actors;
3) Capture interconnect information (relations).

For the first operation, the script parses through the XML document of the Ptolemy structural model and captures the name of the director used. For illustrative purposes, Figure 5.10 is the model used to demonstrate the use of all three Perl scripts for the transformation methodology. The first Perl script finds that the director for Figure 5.10 is the Discrete-Event (DE) director. Next, the script executes the second operation and finds that the actors needed for SystemC translation are *Ramp1*, *Ramp2*, and *AddSubtract*. The script is designed such that it ignores all source and sink actors. A source actor provides input stimulus to either a single or group of actors while a sink actor records the output response of either a single or group of actors [2]. Specifically, the actors *Input_Stimulus* and *Output_Display* are ignored. The reason why these actors are ignored is because these actors are considered testbench components. It is the responsibility of the designer, or design team, to ensure that a Ptolemy structural model is properly tested and verified for correctness. This same principle applies to SystemC models. The third and final operation parses through the XML document of the Ptolemy structural model and captures the names of the interconnects in the Ptolemy structural model. There are four relations in total for this model. Relations *relation2* and *relations3* are obvious when viewing the figure. Although not labeled in the figure, the other two relations are *relation* and *relation4*. Specifically, *relation* is between the *Input_Stimulus* and *Ramp* actors while *relation4* is between the *AddSubtract* and *Output_Display* actors.

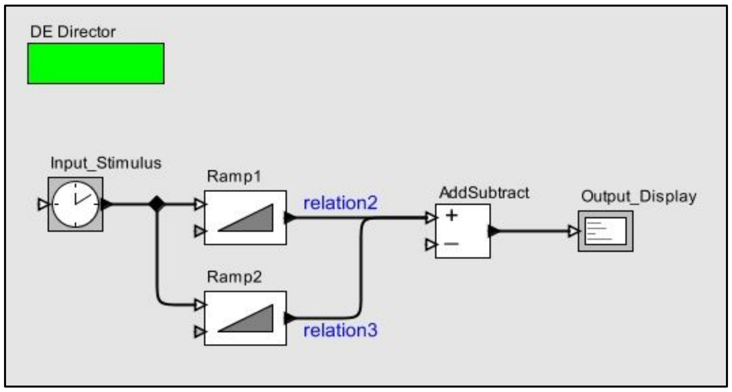
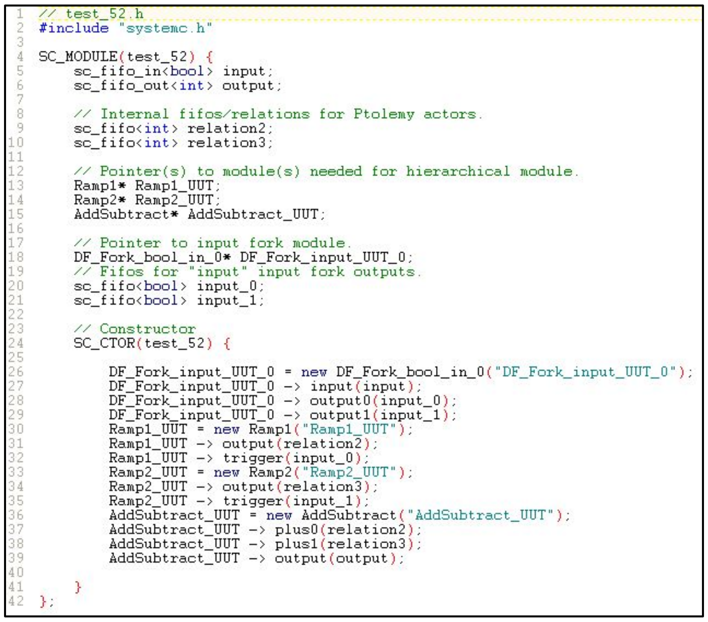


Figure 5.10. *Vergil* window for AddSubtract/Ramp Model

5.3.2 Task 2: Creation of SystemC hierarchical module

This next task of the Perl script produces a wrapper code around the Ptolemy actors found in the second operation of Task 1. This wrapper code is a SystemC module that consists solely of instances of one or more SystemC modules along with internal channels [4]. This wrapper code is referred to as a hierarchical module. Figure 5.11 shows the SystemC hierarchical module of the Ptolemy structural model. In the place of the source and sink actors, the Perl script prompts the designer for I/O names. Specifically, for a source actor, the script prompts the designer for an input port name while, for a sink actor, the script prompts for an output port name. In this figure, the input port name was chosen as *input* while the output port name was chosen as *output*.

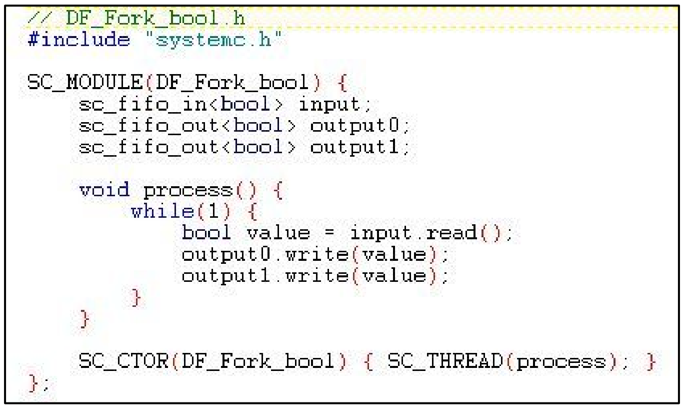


```
// test_52.h
#include "systemc.h"

SC_MODULE(test_52) {
    sc_fifo_in<bool> input;
    sc_fifo_out<int> output;

    // Internal fifos/relations for Ptolemy actors.
    sc_fifo<int> relation2;
    sc_fifo<int> relation3;

    // Pointer(s) to module(s) needed for hierarchical module.
    Ramp1* Ramp1_UUT;
    Ramp2* Ramp2_UUT;
    AddSubtract* AddSubtract_UUT;

    // Pointer to input fork module.
    DF_Fork_bool_in_0* DF_Fork_input_UUT_0;
    // Fifos for "input" input fork outputs.
    sc_fifo<bool> input_0;
    sc_fifo<bool> input_1;

    // Constructor
    SC_CTOR(test_52) {

        DF_Fork_input_UUT_0 = new DF_Fork_bool_in_0("DF_Fork_input_UUT_0");
        DF_Fork_input_UUT_0 -> input(input);
        DF_Fork_input_UUT_0 -> output0(input_0);
        DF_Fork_input_UUT_0 -> output1(input_1);
        Ramp1_UUT = new Ramp1("Ramp1_UUT");
        Ramp1_UUT -> output(relation2);
        Ramp1_UUT -> trigger(input_0);
        Ramp2_UUT = new Ramp2("Ramp2_UUT");
        Ramp2_UUT -> output(relation3);
        Ramp2_UUT -> trigger(input_1);
        AddSubtract_UUT = new AddSubtract("AddSubtract_UUT");
        AddSubtract_UUT -> plus0(relation2);
        AddSubtract_UUT -> plus1(relation3);
        AddSubtract_UUT -> output(output);

    }
};
```

Figure 5.11. SystemC hierarchical module of AddSubtract/Ramp Test Model

5.3.3 Task 2: Insertion of SystemC forking modules and associated FIFOs

Because of the use of KPN MOC in the initial SystemC model, data are represented as tokens. If the token is needed as input to more than one module, that data stream has to be split. To split the token, a fork module is needed [4]. Fork modules split a token and produce as many identical tokens as needed. Figure 5.12 shows the SystemC header file of a fork module used to produce two identical tokens of Boolean data type.



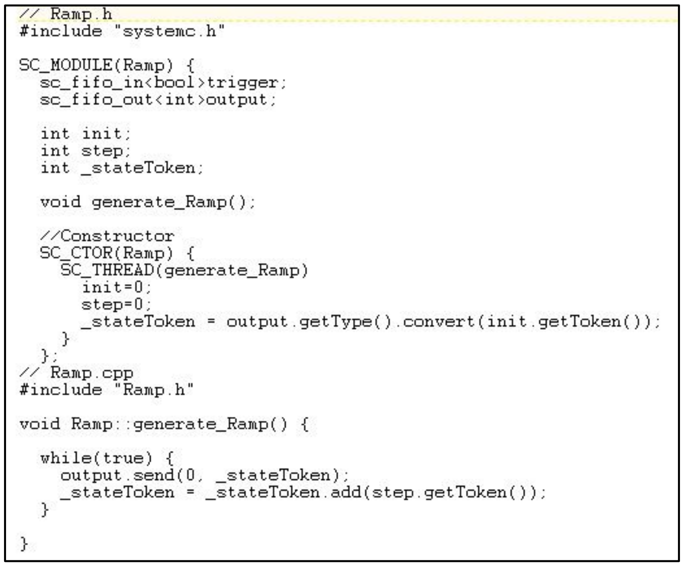
```
// DF_Fork_bool.h
#include "systemc.h"

SC_MODULE(DF_Fork_bool) {
    sc_fifo_in<bool> input;
    sc_fifo_out<bool> output0;
    sc_fifo_out<bool> output1;

    void process() {
        while(1) {
            bool value = input.read();
            output0.write(value);
            output1.write(value);
        }
    }

    SC_CTOR(DF_Fork_bool) { SC_THREAD(process); }
};
```

Figure 5.12. SystemC fork module for creating two identical Boolean tokens

Under the KPN MOC, the *sc_fifo* construct produces a token that has a one-to-one nature in the sense that a token can be the input to *only* one other module. Because the input FIFO named *input* is required by both Ramp1 and Ramp2 SystemC modules, *input* needs to be split. Lines 18 through 21 in Figure 5.11 show a fork module named *DF_Fork_bool_in_0* is used along with the associated FIFO names of *input_0* and *input_1*. The instantiated module is shown on lines 26 through 29. Note that it is the forked FIFOs that are used as input to Ramp1 and Ramp2 modules as can be seen in lines 32 and 35.

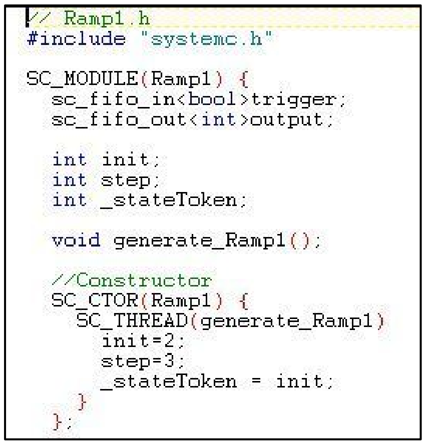
5.3.4 Task 3: Creation of SystemC modules of Ptolemy actors

The Perl script uses the SystemC module templates for the *Ramp* and *AddSubtract* actors. The module template for the *Ramp1* and *Ramp2* actors is shown in Figure 5.13. The syntax/semantics translation is then performed and the results are the SystemC header and source files for each Ramp module. Figure 5.14 shows the Level 2 model of Ramp1 module. Ramp2 module is not shown simply because these modules are nearly identical. Because the Perl script found that these actors require the Ramp SystemC module template, the script performs the additional function of replacing the default parameter values of zero for the parameters *init* and *step* found in each constructor of the Ramp modules. The values of 2 and 3 are the parsed parameter values for the parameters *init* and *step*, respectively, found for *Ramp1* in the Ptolemy structural model, and are shown in the constructor of the header file (e.g.: Ramp1.h) in Figure 5.14. For the *Ramp2* actor, the values of 3 and 5 are the parsed values and are operated on by the Perl script in the same manner as was the *Ramp1* actor. The *AddSubtract* actor has a corresponding module template (see Figure 5.6). Figure 5.7 shows the SystemC header and source files for this module. For verification, Figure 5.15 shows the identical matching of simulation results when comparing the Ptolemy model to the SystemC model.

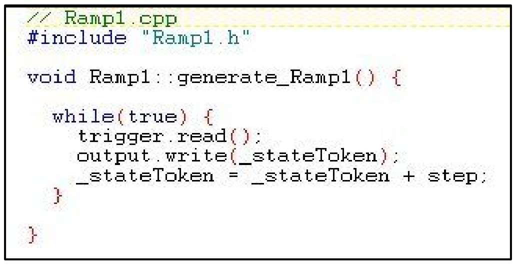


```
// Ramp.h
#include "systemc.h"

SC_MODULE(Ramp) {
  sc_fifo_in<bool>trigger;
  sc_fifo_out<int>output;

  int init;
  int step;
  int _stateToken;

  void generate_Ramp();

  //Constructor
  SC_CTOR(Ramp) {
    SC_THREAD(generate_Ramp)
      init=0;
      step=0;
      _stateToken = output.getType().convert(init.getToken());
    }
  };
// Ramp.cpp
#include "Ramp.h"

void Ramp::generate_Ramp() {

  while(true) {
    output.send(0, _stateToken);
    _stateToken = _stateToken.add(step.getToken());
  }

}
```

Figure 5.13. SystemC module template of Ptolemy *Ramp* actor with 1 input

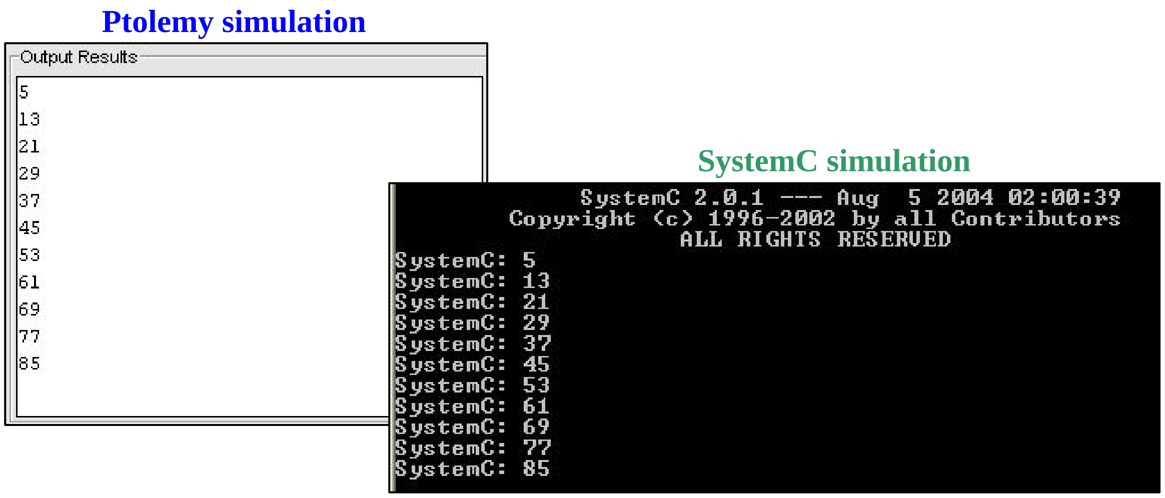


```
// Ramp1.h
#include "systemc.h"

SC_MODULE(Ramp1) {
  sc_fifo_in<bool>trigger;
  sc_fifo_out<int>output;

  int init;
  int step;
  int _stateToken;

  void generate_Ramp1();

  //Constructor
  SC_CTOR(Ramp1) {
    SC_THREAD(generate_Ramp1)
      init=2;
      step=3;
      _stateToken = init;
    }
  };
```



```
// Ramp1.cpp
#include "Ramp1.h"

void Ramp1::generate_Ramp1() {

  while(true) {
    trigger.read();
    output.write(_stateToken);
    _stateToken = _stateToken + step;
  }

}
```

Figure 5.14. SystemC header and source files of Ramp1 module (Level 2)



```
5
13
21
29
37
45
53
61
69
77
85
```

```
          SystemC 2.0.1 --- Aug  5 2004 02:00:39
     Copyright (c) 1996-2002 by all Contributors
                ALL RIGHTS RESERVED
SystemC: 5
SystemC: 13
SystemC: 21
SystemC: 29
SystemC: 37
SystemC: 45
SystemC: 53
SystemC: 61
SystemC: 69
SystemC: 77
SystemC: 85
```

Figure 5.15. Ptolemy and SystemC simulation results of AddSubtract/Ramp Test Model

In reference to the Ptolemy model in Figure 5.10, the Perl script translated the *Ramp* actors and produced the SystemC header and source files. It is important to note an important feature of the first Perl script when it performs Task 3. This important feature deals with the script's ability to parse and gather connective information between Ptolemy actors. For example, when observing each *Ramp* actor in Figure 5.10, it is shown that there are 2 input ports for each. Specifically, these input ports are named *trigger* and *step*. Of the two ports, the *trigger* ports are the only input ports connected to the source actor *Input_Stimulus*. Depending on a designer's choice when using *Vergil*, either one or both input ports may be connected to other Ptolemy actors. The 1-input template is shown in Figure 5.13 of this chapter while the 2-input template is shown in Figure 4.13 of Chapter 4. Because this particular actor has two potential connective configurations for its inputs, two SystemC module templates are created that are coded differently. But although coded differently, the ramp functionality between the two templates remain the same. For Figure 5.10, the 1-input template was used for the Ptolemy *Ramp* actors. The designer does not have to worry if the wrong template will be chosen during a translation because the script is able to decide which of the two Ramp module templates is required when performing the conversion from Ptolemy actor to SystemC module.

5.3.5 Additional Features on Level 1 to Level 2 Translation

As a feature, the Perl script is designed to handle exceptions that may come about when parsing through the XML document of a Ptolemy structural model. As stated in section 5.3.1, the script ignores source actors during translation. The only exceptions are the *Ramp* and *Const* actors [2]. In this research, the *Ramp* actor is considered a non-source actor solely because the ramp is considered a mathematical function. In Ptolemy, the *Const* actor may or may not be considered a source actor depending how it is configured in a Ptolemy structural model. When the Perl script parses through the XML document and encounters a *Const* actor, it makes a determination. If the input port of a *Const* actor is connected to another actor, the *Const* actor is not considered a source actor and, as a result, that actor is translated into a corresponding SystemC module. If the input port of the *Const* actor is not connected to any other actor, then the actor is considered a source actor and is not translated.

Another feature of the Perl script is that, for this research, it only considers either Boolean or integer data types. In the case a double data type is encountered in the Ptolemy I/O of an actor, the script considers this as an integer data type. This leads to a discussion of an important feature of Ptolemy. Before a model is executed through *Vergil*, the I/O data types of the Ptolemy actors are unknown and unresolved. Ptolemy performs a data type resolution on all actors in a model for compatibility from actor to actor [2]. Assuming there are no data type compatibility errors to resolve, the model will successfully execute. The XML document of a Ptolemy structural model does not contain data type information for the I/Os of Ptolemy actors. Because of this, the Perl script must be provided actor data type information elsewhere. The Java file *TypedCompositeActor.java* provides this information by performing a data type resolution check. This file is one of the many Java files in the Ptolemy Java package that allows Ptolemy to represent working models at a high level of abstraction. This Java file was modified to automatically store the resolved actor data type information to a text file. This text file is named *ptolemy_datatypes.txt* and is used by the script to insert the correct data type to the appropriate SystemC module I/O.

Lastly, throughout this research, the Perl scripts bind all SystemC module instantiations in the hierarchical module with the *named form* port binding scheme [4].

5.4 Level 2 to Level 3 Translation

The second Perl script takes as input the initial SystemC source and header files of the Level 2 model and introduces synthesizable SystemC constructs. This transformation consists of replacing the KPN MOC with the RTL MOC. This Perl script performs this MOC transformation to produce a Level 3 SystemC model by executing four tasks. In addition, the script produces input clock and reset ports.

5.4.1 Task 1: Redefining I/O port types of SystemC modules

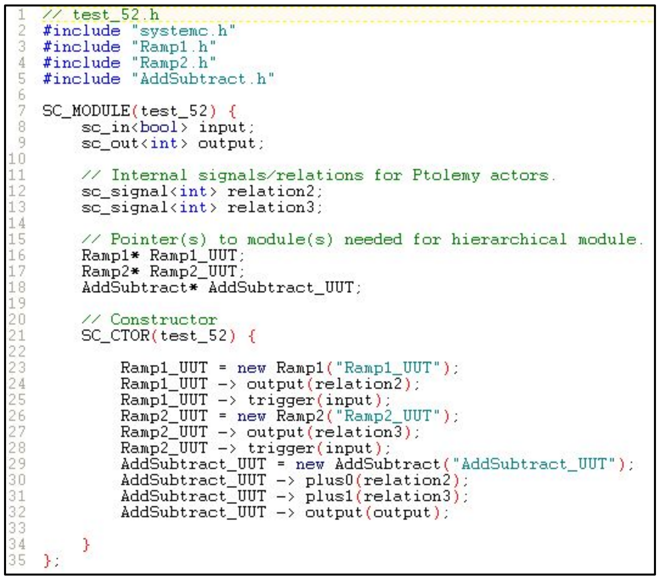
In this task, the Perl script performs the simple and initial task of replacing the I/O port types found in SystemC header files. Level 2 uses *sc_fifo* constructs and, as a result, the I/O port types are *sc_fifo_in* and *sc_fifo_out*. These port types are respectively replaced by the *sc_in* and *sc_out* port types. For example, Table 5.1 shows the Level 2 to Level 3 transformation of I/O port types for the SystemC header files.

Table 5.1. I/O port type transformation

Module Name	Level 2	Level 3
Ramp1	sc_fifo_in<bool> trigger	sc_in<bool> trigger
	sc_fifo_out<int> output	sc_out<int> output
Ramp2	sc_fifo_in<bool> trigger	sc_in<bool> trigger
	sc_fifo_out<int> output	sc_out<int> output
AddSubtract	sc_fifo_in<int> plus0	sc_in<int> plus0
	sc_fifo_in<int> plus1	sc_in<int> plus1
	sc_fifo_out<int> output	sc_out<int> output

5.4.2 Task 2: Removal of SystemC forking modules and associated FIFOs

When the Perl script encounters a fork module during its parsing stage, it removes the fork module along with its associated FIFOs. If the script encounters additional FIFOs that are not associated with fork modules, the script replaces these *sc_fifo* references with *sc_signal* references. An example of the result of this task is Figure 5.16. For example, the *sc_fifo*-to-*sc_signal* replacements are shown when comparing lines 9 and 10 in Figure 5.11 with lines 12 and 13 in Figure 5.16. In addition, the FIFOs of *input_0* and *input_1* on lines 32 and 35 in Figure 5.11 are replaced with the signal of *input* in lines 25 and 28 in Figure 5.16.

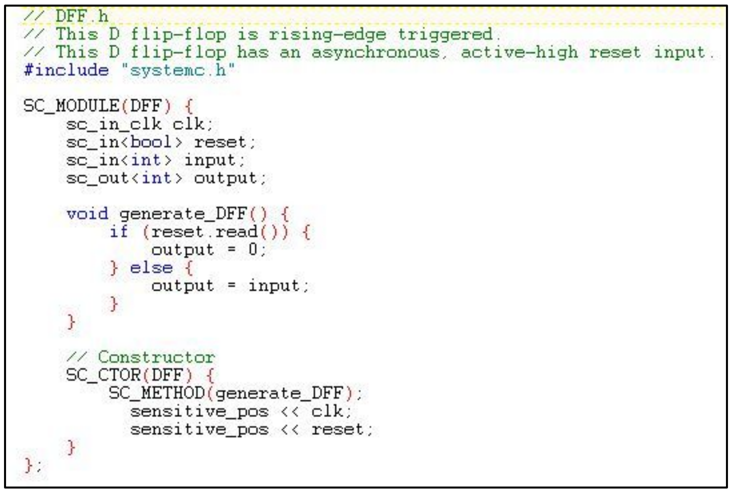


```
// test_52.h
#include "systemc.h"
#include "Ramp1.h"
#include "Ramp2.h"
#include "AddSubtract.h"

SC_MODULE(test_52) {
    sc_in<bool> input;
    sc_out<int> output;

    // Internal signals/relations for Ptolemy actors.
    sc_signal<int> relation2;
    sc_signal<int> relation3;

    // Pointer(s) to module(s) needed for hierarchical module.
    Ramp1* Ramp1_UUT;
    Ramp2* Ramp2_UUT;
    AddSubtract* AddSubtract_UUT;

    // Constructor
    SC_CTOR(test_52) {

        Ramp1_UUT = new Ramp1("Ramp1_UUT");
        Ramp1_UUT -> output(relation2);
        Ramp1_UUT -> trigger(input);
        Ramp2_UUT = new Ramp2("Ramp2_UUT");
        Ramp2_UUT -> output(relation3);
        Ramp2_UUT -> trigger(input);
        AddSubtract_UUT = new AddSubtract("AddSubtract_UUT");
        AddSubtract_UUT -> plus0(relation2);
        AddSubtract_UUT -> plus1(relation3);
        AddSubtract_UUT -> output(output);

    }
};
```

Figure 5.16. First intermediate SystemC hierarchical module using RTL MOC

5.4.3 Task 3: Insertion of D-type flip-flops and clock and reset ports

To implement the RTL structure shown in Figure 5.9, the script performs the function of inserting D-type flip-flops by generally following the three steps in section 5.2. In addition, the script introduces additional signals associated with the I/O ports of the SystemC wrapper code (hierarchical module) named in section 5.3.2 (e.g.: *input* and *output*). This research chose the convention of including the suffix *_reg* to those additional signals. Figure 5.17 illustrates a SystemC header file of a D-type flip-flop with an active-high, asynchronous reset port.



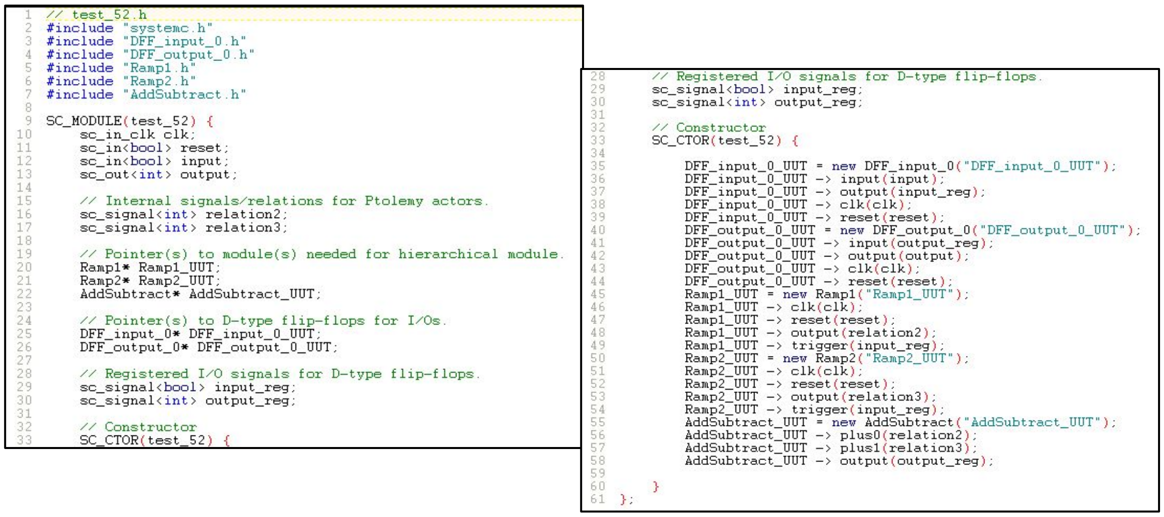
```
// DFF.h
// This D flip-flop is rising-edge triggered.
// This D flip-flop has an asynchronous, active-high reset input.
#include "systemc.h"

SC_MODULE(DFF) {
    sc_in_clk clk;
    sc_in<bool> reset;
    sc_in<int> input;
    sc_out<int> output;

    void generate_DFF() {
        if (reset.read()) {
            output = 0;
        } else {
            output = input;
        }
    }

    // Constructor
    SC_CTOR(DFF) {
        SC_METHOD(generate_DFF);
          sensitive_pos << clk;
          sensitive_pos << reset;
    }
};
```

Figure 5.17. SystemC module of rising edge triggered D-type flip-flop with asynchronous, active-high reset

Following the SystemC hierarchical module shown in Figure 5.16, the Perl script inserts two flip-flops. One is for the input port *input* while the other is for the output port *output*. Generally, the Perl script implements Task 3 by performing three operations as follows:

1) Insert clock and reset input ports to hierarchical module and, if necessary, module instantiations;
2) Insert the appropriate number of pointers for appropriate number of flip-flops;
3) Insert I/O signals with *_reg* suffix.

As can be seen in Figure 5.18, the Perl script completes the task of translating the Level 2 model to a Level 3 model. The RTL structure of this Level 3 model is shown in Figure 5.19. Note, the three steps outlined in section 5.2 take the Ptolemy model in Figure 5.10 and produce this translated RTL structure.



```
// test_52.h
#include "systemc.h"
#include "DFF_input_0.h"
#include "DFF_output_0.h"
#include "Ramp1.h"
#include "Ramp2.h"
#include "AddSubtract.h"

SC_MODULE(test_52) {
    sc_in_clk clk;
    sc_in<bool> reset;
    sc_in<bool> input;
    sc_out<int> output;

    // Internal signals/relations for Ptolemy actors.
    sc_signal<int> relation2;
    sc_signal<int> relation3;

    // Pointer(s) to module(s) needed for hierarchical module.
    Ramp1* Ramp1_UUT;
    Ramp2* Ramp2_UUT;
    AddSubtract* AddSubtract_UUT;

    // Pointer(s) to D-type flip-flops for I/Os.
    DFF_input_0* DFF_input_0_UUT;
    DFF_output_0* DFF_output_0_UUT;

    // Registered I/O signals for D-type flip-flops.
    sc_signal<bool> input_reg;
    sc_signal<int> output_reg;

    // Constructor
    SC_CTOR(test_52) {
```

```
    // Registered I/O signals for D-type flip-flops.
    sc_signal<bool> input_reg;
    sc_signal<int> output_reg;

    // Constructor
    SC_CTOR(test_52) {

        DFF_input_0_UUT = new DFF_input_0("DFF_input_0_UUT");
        DFF_input_0_UUT -> input(input);
        DFF_input_0_UUT -> output(input_reg);
        DFF_input_0_UUT -> clk(clk);
        DFF_input_0_UUT -> reset(reset);
        DFF_output_0_UUT = new DFF_output_0("DFF_output_0_UUT");
        DFF_output_0_UUT -> input(output_reg);
        DFF_output_0_UUT -> output(output);
        DFF_output_0_UUT -> clk(clk);
        DFF_output_0_UUT -> reset(reset);
        Ramp1_UUT = new Ramp1("Ramp1_UUT");
        Ramp1_UUT -> clk(clk);
        Ramp1_UUT -> reset(reset);
        Ramp1_UUT -> output(relation2);
        Ramp1_UUT -> trigger(input_reg);
        Ramp2_UUT = new Ramp2("Ramp2_UUT");
        Ramp2_UUT -> clk(clk);
        Ramp2_UUT -> reset(reset);
        Ramp2_UUT -> output(relation3);
        Ramp2_UUT -> trigger(input_reg);
        AddSubtract_UUT = new AddSubtract("AddSubtract_UUT");
        AddSubtract_UUT -> plus0(relation2);
        AddSubtract_UUT -> plus1(relation3);
        AddSubtract_UUT -> output(output_reg);

    }
};
```

Figure 5.18. SystemC hierarchical module under RTL modeling (Level 3)

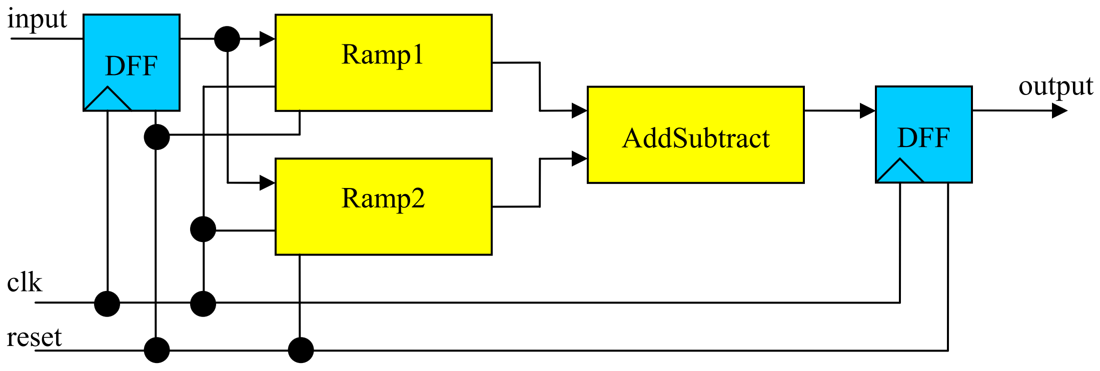


Figure 5.19. RTL structure of AddSubtract/Ramp Test Model

5.4.4 Task 4: Translating KPN MOC to RTL MOC of SystemC modules

The first three tasks dealt with transforming the SystemC hierarchical module. Task 4 deals with doing the same but only to modules indicated as pointers in the hierarchical module. Figure 5.5 shows RTL MOC transformation of the Ramp modules. Figure 5.8 shows a similar RTL MOC transformation for the AddSubtract module.

Figure 5.15 shows the simulation results of both Level 1 and Level 2 models. The Ptolemy simulation results are from the Level 1 model (see Figure 5.10) while the SystemC simulation results are from the Level 2 model using the KPN model of computation. Figure 5.20 shows the graphical results under the RTL model of computation for the Level 3 model. As can be seen by the results, *SystemC.output[31:0]* matches the benchmark results of the Ptolemy simulation results in Figure 5.15.

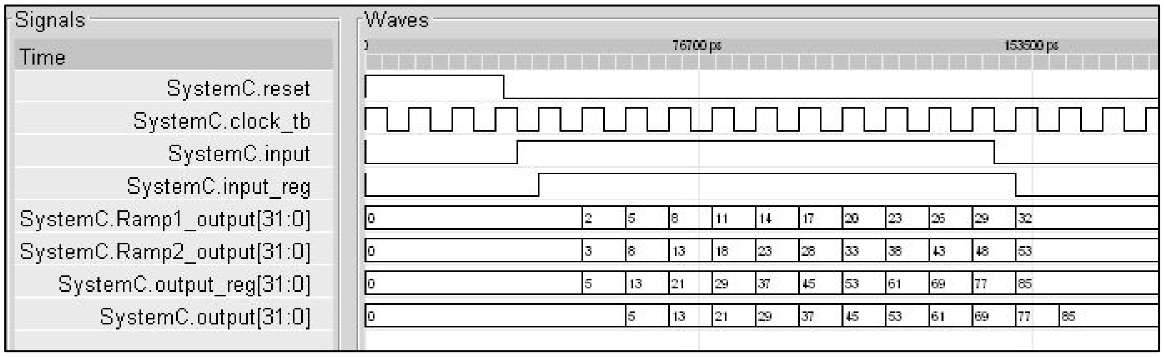
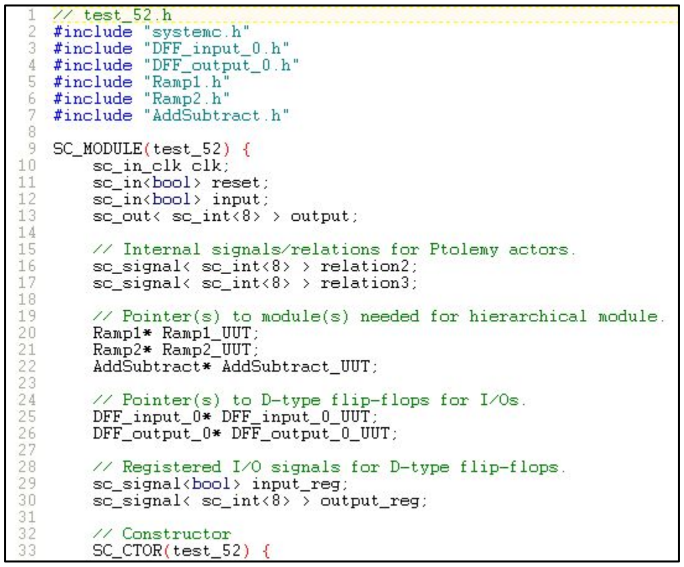


Figure 5.20. Simulation of Level 3 model of AddSubtract/Ramp Test Model

5.5 Level 3 to Level 4 Translation

The third and final Perl script performs the task of constraining the binary bit length of I/O ports and internal variables of a Level 3 SystemC modules and producing a Level 4 model. This falls under the principles given in section 2.2.1 where a designer makes hardware considerations when moving down the 5-level design process. In this research, the convention of making the bit length of all I/O ports and internal variables of an entire SystemC model the same length is chosen. The uniform bit length of the Level 4 model used in this illustration is eight. The Perl script takes as input the Level 3 model and introduces this data length constraint. Figure 5.21 shows the result of this transformation for the SystemC hierarchical module (see Figure 5.18). Notice the unconstrained-to-constrained bit length transformation when comparing Figures 5.18 and 5.21. Specifically, lines 13, 16, 17, and 30 of each respective model show the transformation performed by the Perl script.



```
// test_52.h
#include "systemc.h"
#include "DFF_input_0.h"
#include "DFF_output_0.h"
#include "Ramp1.h"
#include "Ramp2.h"
#include "AddSubtract.h"

SC_MODULE(test_52) {
    sc_in_clk clk;
    sc_in<bool> reset;
    sc_in<bool> input;
    sc_out< sc_int<8> > output;

    // Internal signals/relations for Ptolemy actors.
    sc_signal< sc_int<8> > relation2;
    sc_signal< sc_int<8> > relation3;

    // Pointer(s) to module(s) needed for hierarchical module.
    Ramp1* Ramp1_UUT;
    Ramp2* Ramp2_UUT;
    AddSubtract* AddSubtract_UUT;

    // Pointer(s) to D-type flip-flops for I/Os.
    DFF_input_0* DFF_input_0_UUT;
    DFF_output_0* DFF_output_0_UUT;

    // Registered I/O signals for D-type flip-flops.
    sc_signal<bool> input_reg;
    sc_signal< sc_int<8> > output_reg;

    // Constructor
    SC_CTOR(test_52) {
```

Figure 5.21. SystemC hierarchical module with 8-bit data constraints (Level 4)

The translation goal of the Perl scripts is achieved for this example. This Level 4 model is the synthesis-ready SystemC model. This model is used as input to synthesis tools to produce the synthesized model of Level 5 (see Table 2.2). Section 5.6 illustrates the synthesized results of the SystemC Compiler discussed in section 2.1.2 in terms of simulation verification.

5.6 Post-synthesis Results of Level 5 Model

The final part of the transformation methodology is to produce a synthesized, gate-level (Level 5) model. The SystemC header and source files of the Level 4 model are the inputs to the synthesis tool SystemC Compiler as stated in the end of the preceding section. The designer is responsible for synthesizing the overall Level 4 model to produce either an ASIC or FPGA (Level 5) model. For the Level 4 model shown in Figure 5.21, the Level 5 model is an ASIC. Figure 5.22 shows the synthesized simulation results. As can be seen by the results, this figure matches the simulation results shown in Figure 5.20.

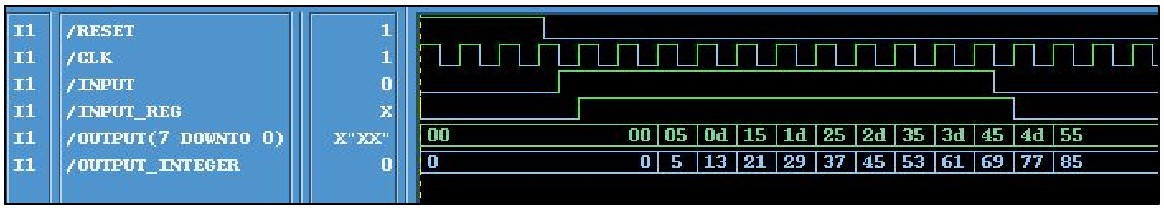


Figure 5.22. Post-synthesis simulation of ASIC (Level 5) model

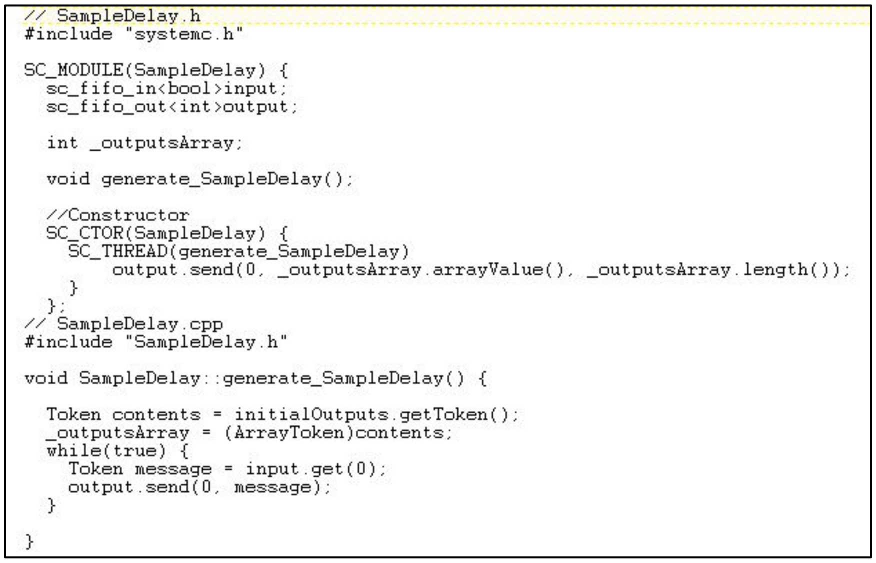
Section 3.6 stated that, in the ASIC/FPGA design industry, a one-to-many mapping of components is common practice. The above Level 5 model is an example of this mapping type. Table 5.2 shows the one-to-many mapping of components when moving from Level 4 to Level 5 of the 5-level design process. The "Level 4 Component Count" column is always an entry of one for each SystemC module since this is a one-to-many mapping. Note the considerable component count of both Ramp modules under the "Level 5 Component Count" column. The transformation to a gate-level circuit is done by a proprietary commercial synthesis algorithm and is thus not the focus of this research.

Table 5.2. One-to-many mapping of Level 4 to Level 5 translation

Module Name	Level 4 Component Count	Level 5 Component Count
DFF_input_0	1	2
DFF_output_0	1	9
Ramp1	1	48
Ramp2	1	48
AddSubtract	1	9

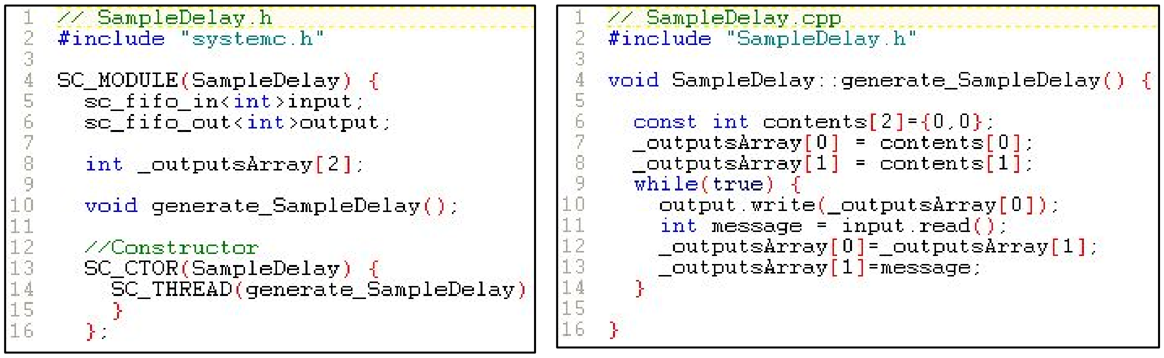
5.7 Module Template and Levels 2 through 4 Representation of Ptolemy *SampleDelay* Actor

As stated in section 5.1.2, the module template and Level 2 model of the Ptolemy *SampleDelay* actor shown in Figure 5.2a are to be presented in this chapter. This section shows these template and model along with the Level 3 model for this actor. Figures 5.23 through 5.25 illustrate these documents. Note that this actor falls under the *with delay* category defined in section 5.1.2, and, as a result, there is control logic for both Levels 2 and 3 when compared to the module template. Sections 5.7.1 and 5.7.2 explain how the delay is achieved for this actor in SystemC for both levels.



```
// SampleDelay.h
#include "systemc.h"

SC_MODULE(SampleDelay) {
  sc_fifo_in<bool>input;
  sc_fifo_out<int>output;

  int _outputsArray;

  void generate_SampleDelay();

  //Constructor
  SC_CTOR(SampleDelay) {
    SC_THREAD(generate_SampleDelay)
        output.send(0, _outputsArray.arrayValue(), _outputsArray.length());
    }
  };
// SampleDelay.cpp
#include "SampleDelay.h"

void SampleDelay::generate_SampleDelay() {

  Token contents = initialOutputs.getToken();
  _outputsArray = (ArrayToken)contents;
  while(true) {
    Token message = input.get(0);
    output.send(0, message);
  }

}
```

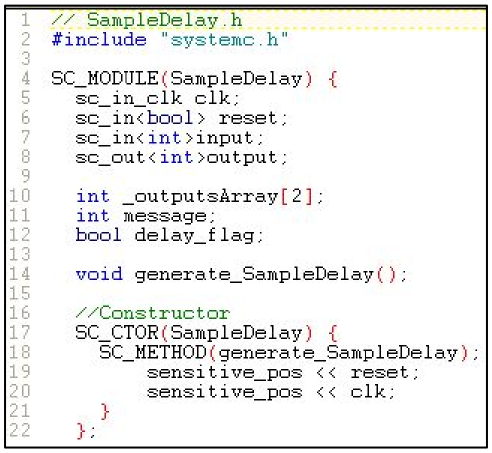
Figure 5.23. SystemC module template of Ptolemy *SampleDelay* actor



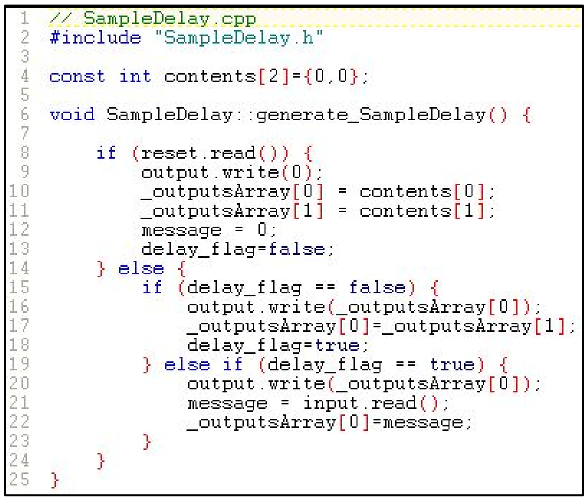
```
// SampleDelay.h
#include "systemc.h"

SC_MODULE(SampleDelay) {
  sc_fifo_in<int>input;
  sc_fifo_out<int>output;

  int _outputsArray[2];

  void generate_SampleDelay();

  //Constructor
  SC_CTOR(SampleDelay) {
    SC_THREAD(generate_SampleDelay)
    }
  };
```

```
// SampleDelay.cpp
#include "SampleDelay.h"

void SampleDelay::generate_SampleDelay() {

  const int contents[2]={0,0};
  _outputsArray[0] = contents[0];
  _outputsArray[1] = contents[1];
  while(true) {
    output.write(_outputsArray[0]);
    int message = input.read();
    _outputsArray[0]=_outputsArray[1];
    _outputsArray[1]=message;
  }

}
```

Figure 5.24. Level 2 model of Ptolemy *SampleDelay* actor using KPN MOC



```
// SampleDelay.h
#include "systemc.h"

SC_MODULE(SampleDelay) {
  sc_in_clk clk;
  sc_in<bool> reset;
  sc_in<int>input;
  sc_out<int>output;

  int _outputsArray[2];
  int message;
  bool delay_flag;

  void generate_SampleDelay();

  //Constructor
  SC_CTOR(SampleDelay) {
    SC_METHOD(generate_SampleDelay);
        sensitive_pos << reset;
        sensitive_pos << clk;
    }
  };
```



```
// SampleDelay.cpp
#include "SampleDelay.h"

const int contents[2]={0,0};

void SampleDelay::generate_SampleDelay() {

    if (reset.read()) {
        output.write(0);
        _outputsArray[0] = contents[0];
        _outputsArray[1] = contents[1];
        message = 0;
        delay_flag=false;
    } else {
        if (delay_flag == false) {
            output.write(_outputsArray[0]);
            _outputsArray[0]=_outputsArray[1];
            delay_flag=true;
        } else if (delay_flag == true) {
            output.write(_outputsArray[0]);
            message = input.read();
            _outputsArray[0]=message;
        }
    }
}
```

Figure 5.25. Level 3 model of Ptolemy *SampleDelay* actor using RTL MOC

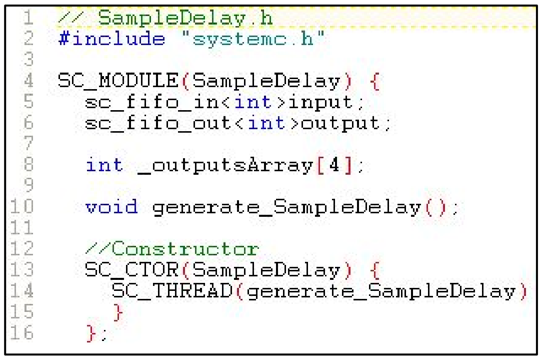
5.7.1 Achieving Delay in Level 2 model of the SampleDelay module

To emulate the delay behavior of the Ptolemy *SampleDelay* actor at the SystemC level, a method of data storage is required in which data stored last will be recovered last. The use of arrays is the data storage method chosen in this research. Specifically, an array functions as a shift register following the FIFO (First-In, First-Out) method. For both Levels 2 and 3, this shift register is called *_outputsArray* and is shown in the header files of Figures 5.24 and 5.25 (left file in both figures).

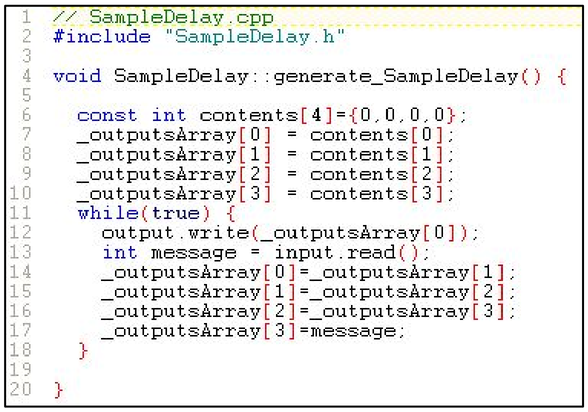
For the Level 2 model shown in Figure 5.24, the SampleDelay module is configured to have a delay of two. Hence, the *_outputsArray* array has a length of 2 as shown in line 8 of the header file. In the source file, this array is initialized to another array named *contents*. This initializing array contains the initial values for the module. The while-loop executes the FIFO method by doing the steps in a repeated fashion as follows:

1) Output the content of *_outputsArray[0]* onto *output*;
2) Store the contents of *input* into the variable *message*;
3) Perform the shift function: *_outputsArray*[i] = *_outputsArray*[i+1] where i is 0≤i<|*_outputsArray*|-1;
4) Store the variable *message* into the (|*_outputsArray*|-1)th position of *_outputsArray*.

Specifically, line 10 is Step (1), line 11 is Step (2), line 12 is Step (3), and line 13 is Step (4). To demonstrate the versatility of the above four steps, Figure 5.26 is a SampleDelay module configured to have a delay of 4. As can be seen in the source file of this figure, line 12 is Step (1), line 13 is Step (2), lines 14 through 16 is Step (3), and line 17 is Step (4).



```
1  // SampleDelay.h
2  #include "systemc.h"
3
4  SC_MODULE(SampleDelay) {
5    sc_fifo_in<int>input;
6    sc_fifo_out<int>output;
7
8    int _outputsArray[4];
9
10   void generate_SampleDelay();
11
12   //Constructor
13   SC_CTOR(SampleDelay) {
14     SC_THREAD(generate_SampleDelay)
15     }
16   };
```

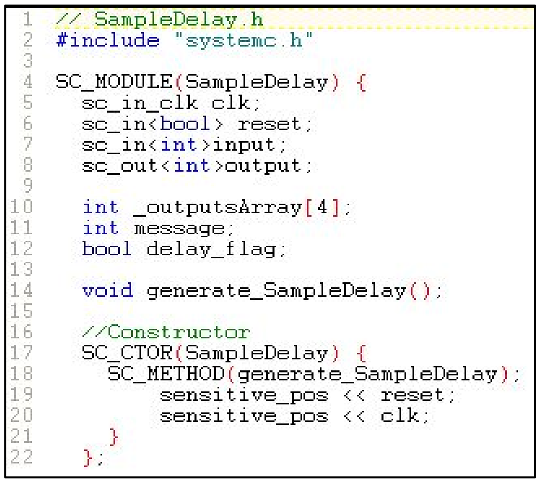


```
1  // SampleDelay.cpp
2  #include "SampleDelay.h"
3
4  void SampleDelay::generate_SampleDelay() {
5
6    const int contents[4]={0,0,0,0};
7    _outputsArray[0] = contents[0];
8    _outputsArray[1] = contents[1];
9    _outputsArray[2] = contents[2];
10   _outputsArray[3] = contents[3];
11   while(true) {
12     output.write(_outputsArray[0]);
13     int message = input.read();
14     _outputsArray[0]=_outputsArray[1];
15     _outputsArray[1]=_outputsArray[2];
16     _outputsArray[2]=_outputsArray[3];
17     _outputsArray[3]=message;
18   }
19
20 }
```

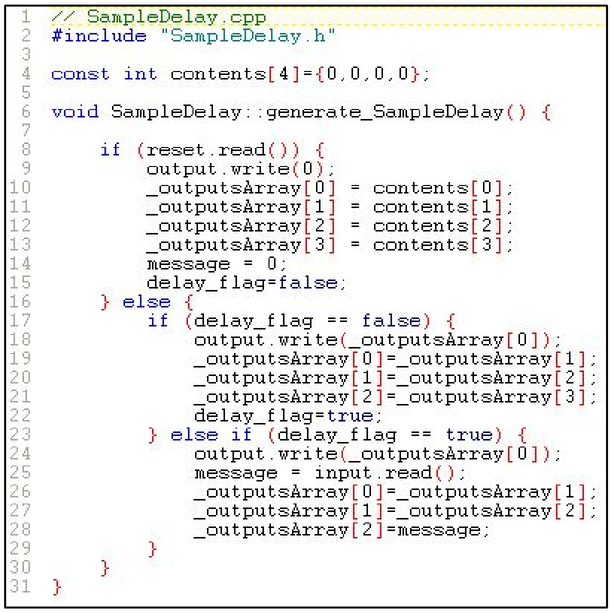
Figure 5.26. Level 2 model of Ptolemy *SampleDelay* actor with delay of 4

5.7.2 Achieving Delay in Level 3 model of the SampleDelay module

For the Level 3 model shown in Figure 5.25, this module is configured for a delay of 2 like the Level 2 model shown in Figure 5.24. Because of RTL modeling, the source file is modified to preserve the delay behavior through the use of the Boolean variable *delay_flag*. During the non-reset condition, when *delay_flag* is false, the module performs the one time operation of shifting its array contents out while not shifting *input* into the (|*_outputsArray*|-1)th position of *_outputsArray*. The variable *delay_flag* is then set to true (at the next rising edge of *clk*) and it is then that the module begins shifting the entire array along with shifting *input* into the (|*_outputsArray*|-1)th position of *_outputsArray*. Like Figure 5.26, Figure 5.27 shows a SampleDelay module configured for a delay of 4.



```
// SampleDelay.h
#include "systemc.h"

SC_MODULE(SampleDelay) {
  sc_in_clk clk;
  sc_in<bool> reset;
  sc_in<int>input;
  sc_out<int>output;

  int _outputsArray[4];
  int message;
  bool delay_flag;

  void generate_SampleDelay();

  //Constructor
  SC_CTOR(SampleDelay) {
    SC_METHOD(generate_SampleDelay);
        sensitive_pos << reset;
        sensitive_pos << clk;
    }
  };
```



```
// SampleDelay.cpp
#include "SampleDelay.h"

const int contents[4]={0,0,0,0};

void SampleDelay::generate_SampleDelay() {

    if (reset.read()) {
        output.write(0);
        _outputsArray[0] = contents[0];
        _outputsArray[1] = contents[1];
        _outputsArray[2] = contents[2];
        _outputsArray[3] = contents[3];
        message = 0;
        delay_flag=false;
    } else {
        if (delay_flag == false) {
            output.write(_outputsArray[0]);
            _outputsArray[0]=_outputsArray[1];
            _outputsArray[1]=_outputsArray[2];
            _outputsArray[2]=_outputsArray[3];
            delay_flag=true;
        } else if (delay_flag == true) {
            output.write(_outputsArray[0]);
            message = input.read();
            _outputsArray[0]=_outputsArray[1];
            _outputsArray[1]=_outputsArray[2];
            _outputsArray[2]=message;
        }
    }
}
```

Figure 5.27. Level 3 model of Ptolemy *SampleDelay* actor with delay of 4

Chapter 6. Effectiveness of the Transformation Methodology

To demonstrate the effectiveness of the transformation methodology, the transformation of two multi-domain Ptolemy models is presented in this chapter. These multi-domain models are heterogeneous structural models. The simulation results of each of these Ptolemy models are benchmarks for validating the lower levels in terms of model behavior. The first heterogeneous model is a producer/consumer model consisting of three Ptolemy domains. The second heterogeneous model consists of four Ptolemy domains, and this model is basically another producer/consumer model. The difference between these two multi-domain models is that the first model contains one producer and one consumer while the second model contains three producers and one consumer which require multiplexing.

Once the transformation methodology produces a Level 4 model for each heterogeneous model, the synthesis tool (SystemC Compiler) uses that Level 4 model as input to produce a synthesized gate-level model. Each gate-level model is then properly verified through gate-level simulation.

In this chapter, sections 6.1 and 6.2 describe the two heterogeneous Ptolemy models, respectively, in detail in terms of actor composition and model behavior.

6.1 HDE-DE-SDF Heterogeneous Ptolemy Model

Figure 6.1 shows the Ptolemy structural model composed of three domains and their corresponding directors and two composite actors. The three domains are Hardware Discrete Event (HDE), Discrete Event (DE), and Synchronous Data Flow (SDF) [2]. A brief definition of each domain is as follows:

1) HDE: This director implements the discrete-event model of computation with memoried signals.
2) DE: This director implements the discrete-event MOC.
3) SDF: This domain is useful for modeling simple dataflow systems without complicated control flow.

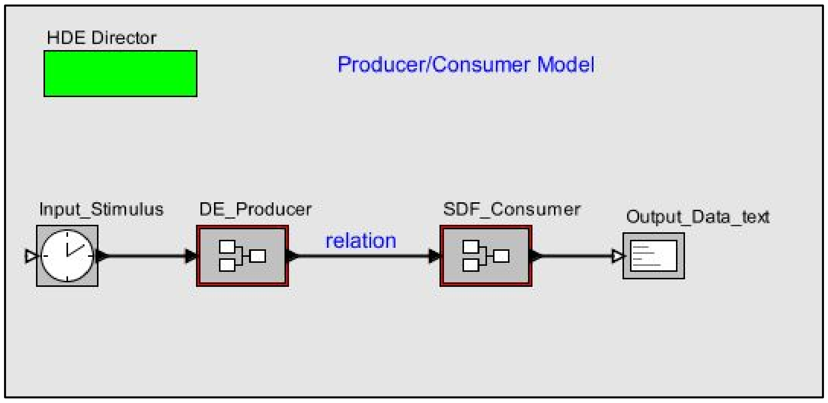


Figure 6.1. HDE-DE-SDF Domain of Producer/Consumer Model

The composite actor *DE_Producer* is composed of three actors as shown in Figure 6.2. These actors are *Const*, *Ramp*, and *ArrayElement* and have the user-defined names of *Data_In*, *Ramp*, and *ArrayElement*, respectively. The purpose of this composite actor is to index each element in the array stored in *Data_In*. As can be seen in Figure 6.2, the array contents is {0,12,3,65,87,43,40,2,8,28,35}. Each indexed element is placed on the output port *data_out*. The *Ramp* actor generates the index while the *ArrayElement* actor accesses the indexed value and places it on its output. Whenever a Boolean true token is received on the input port *port_in*, an indexed value is produced on to *data_out*. All three actors are regulated by the DE director.

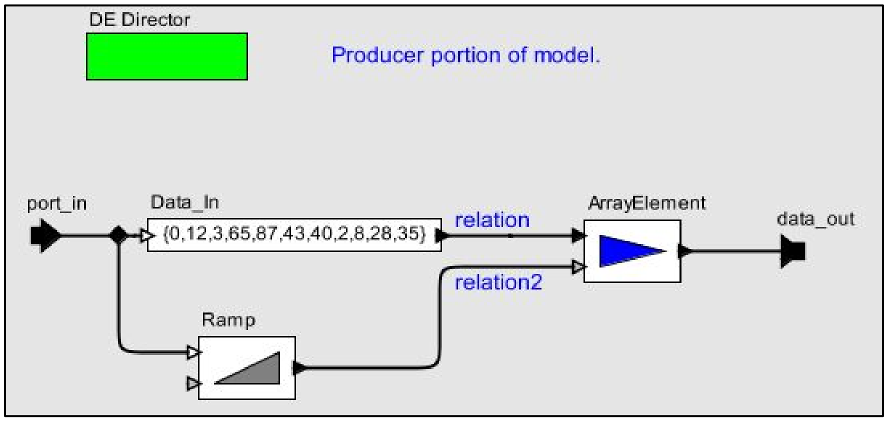


Figure 6.2. Actor composition of *DE_Producer* composite actor

The composite actor *SDF_Consumer* is composed of four actors and is shown in Figure 6.3. These actors are *Const*, *AddSubtract*, *MultiplyDivide*, and *SampleDelay*. The *Const* actor has the user-defined name of *Coefficient*. This composite actor represents a first-order IIR filter [13]. The filter coefficient is an integer value of five. The purpose of this composite actor is to take the integer token on *data_in* as input to the filter and produce an output on to *data_out*. These four actors are controlled by the SDF director.

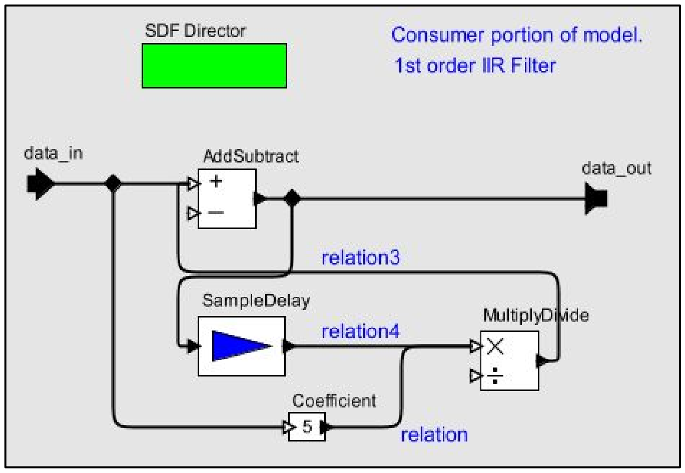
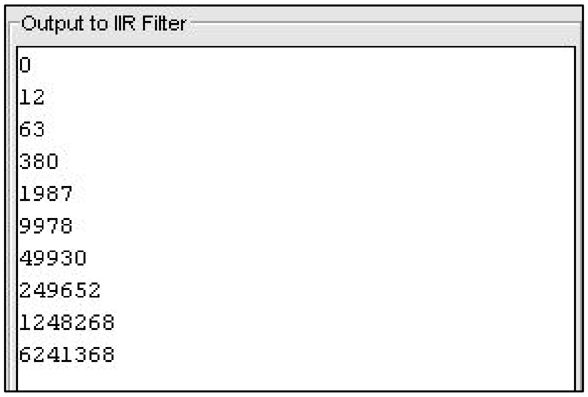


Figure 6.3. Actor composition of *SDF_Consumer* composite actor

Both composite actors are under the HDE director. The composite actor *DE_Producer* produces an integer, and that integer is used as input to *SDF_Consumer* and is consumed/computed by the IIR filter. Figure 6.4 shows the output of the entire model (e.g.: the output of the *SDF_Consumer* composite actor). The source actor *Input_Stimulus* provides the Boolean true tokens while the sink actor *Output_Data_text* displays the output response of the producer/consumer model. These results are as expected and are now the benchmark for verifying Levels 2 through 4.

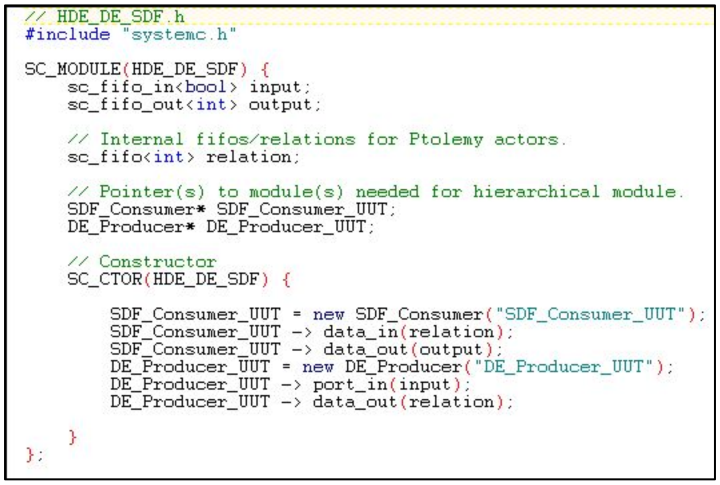


```
0
12
63
380
1987
9978
49930
249652
1248268
6241368
```

Figure 6.4. Output response to Producer/Consumer Model

6.1.1 Level 1 to Level 2 Translation

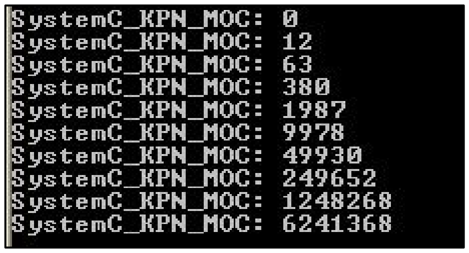
The first Perl script performs the initial level-to-level translation. Figure 6.5 shows the SystemC hierarchical module of Figure 6.1.



```
// HDE_DE_SDF.h
#include "systemc.h"

SC_MODULE(HDE_DE_SDF) {
    sc_fifo_in<bool> input;
    sc_fifo_out<int> output;

    // Internal fifos/relations for Ptolemy actors.
    sc_fifo<int> relation;

    // Pointer(s) to module(s) needed for hierarchical module.
    SDF_Consumer* SDF_Consumer_UUT;
    DE_Producer* DE_Producer_UUT;

    // Constructor
    SC_CTOR(HDE_DE_SDF) {

        SDF_Consumer_UUT = new SDF_Consumer("SDF_Consumer_UUT");
        SDF_Consumer_UUT -> data_in(relation);
        SDF_Consumer_UUT -> data_out(output);
        DE_Producer_UUT = new DE_Producer("DE_Producer_UUT");
        DE_Producer_UUT -> port_in(input);
        DE_Producer_UUT -> data_out(relation);

    }
};
```

Figure 6.5. SystemC hierarchical module of Producer/Consumer model under KPN MOC

The other SystemC header and source files for this Level 2 representation are not shown because of the impracticality of showing large amounts of code. The only SystemC code shown is for the top-level. We take this approach for this example and for the one that follows. Figure 6.6 shows the simulation results for this model under the KPN model of computation. As can be seen by the results, the simulation identically matches the benchmark of Figure 6.4.

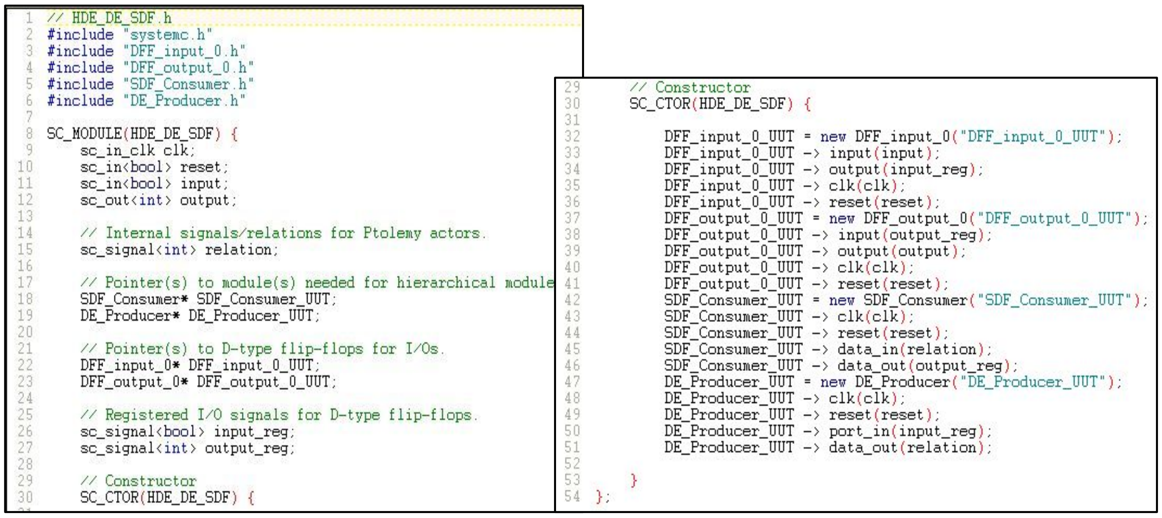


```
SystemC_KPN_MOC: 0
SystemC_KPN_MOC: 12
SystemC_KPN_MOC: 63
SystemC_KPN_MOC: 380
SystemC_KPN_MOC: 1987
SystemC_KPN_MOC: 9978
SystemC_KPN_MOC: 49930
SystemC_KPN_MOC: 249652
SystemC_KPN_MOC: 1248268
SystemC_KPN_MOC: 6241368
```

Figure 6.6. Level 2 (KPN MOC) simulation results for Producer/Consumer model

6.1.2 Level 2 to Level 3 Translation

This second Perl script takes as input the initial SystemC source and header files of the Level 2 model and introduces synthesizable SystemC constructs. Figure 6.7 shows the SystemC hierarchical module under the RTL MOC. The result is a Level 3 representation.



```
// HDE_DE_SDF.h
#include "systemc.h"
#include "DFF_input_0.h"
#include "DFF_output_0.h"
#include "SDF_Consumer.h"
#include "DE_Producer.h"

SC_MODULE(HDE_DE_SDF) {
    sc_in_clk clk;
    sc_in<bool> reset;
    sc_in<bool> input;
    sc_out<int> output;

    // Internal signals/relations for Ptolemy actors.
    sc_signal<int> relation;

    // Pointer(s) to module(s) needed for hierarchical module
    SDF_Consumer* SDF_Consumer_UUT;
    DE_Producer* DE_Producer_UUT;

    // Pointer(s) to D-type flip-flops for I/Os.
    DFF_input_0* DFF_input_0_UUT;
    DFF_output_0* DFF_output_0_UUT;

    // Registered I/O signals for D-type flip-flops.
    sc_signal<bool> input_reg;
    sc_signal<int> output_reg;

    // Constructor
    SC_CTOR(HDE_DE_SDF) {
```

```
    // Constructor
    SC_CTOR(HDE_DE_SDF) {

        DFF_input_0_UUT = new DFF_input_0("DFF_input_0_UUT");
        DFF_input_0_UUT -> input(input);
        DFF_input_0_UUT -> output(input_reg);
        DFF_input_0_UUT -> clk(clk);
        DFF_input_0_UUT -> reset(reset);
        DFF_output_0_UUT = new DFF_output_0("DFF_output_0_UUT");
        DFF_output_0_UUT -> input(output_reg);
        DFF_output_0_UUT -> output(output);
        DFF_output_0_UUT -> clk(clk);
        DFF_output_0_UUT -> reset(reset);
        SDF_Consumer_UUT = new SDF_Consumer("SDF_Consumer_UUT");
        SDF_Consumer_UUT -> clk(clk);
        SDF_Consumer_UUT -> reset(reset);
        SDF_Consumer_UUT -> data_in(relation);
        SDF_Consumer_UUT -> data_out(output_reg);
        DE_Producer_UUT = new DE_Producer("DE_Producer_UUT");
        DE_Producer_UUT -> clk(clk);
        DE_Producer_UUT -> reset(reset);
        DE_Producer_UUT -> port_in(input_reg);
        DE_Producer_UUT -> data_out(relation);

    }
};
```

Figure 6.7. SystemC hierarchical module of Producer/Consumer model under RTL MOC

Figures 6.8a through 6.8c show the simulation results for this model under the RTL model of computation. As can be seen by the results on the output signal *SystemC.output[31:0]*, the simulation identically matches the benchmark of Figure 6.4.

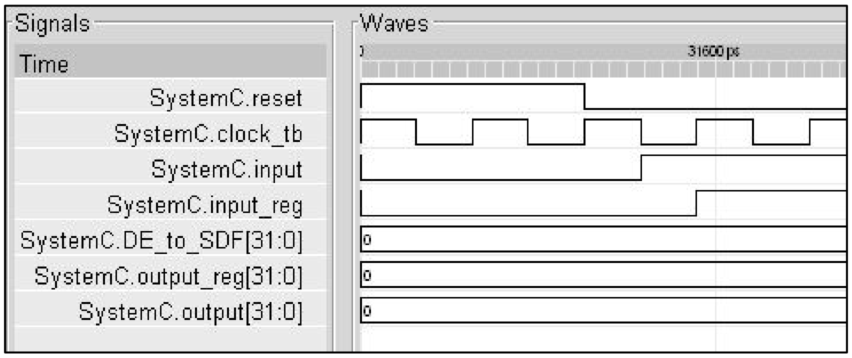


Figure 6.8a. Level 3 (RTL MOC) simulation results for Producer/Consumer model (1/3)

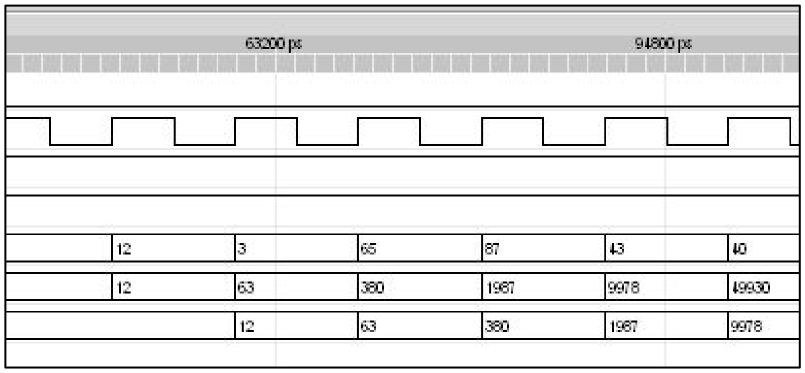


Figure 6.8b. Level 3 (RTL MOC) simulation results for Producer/Consumer model (2/3)

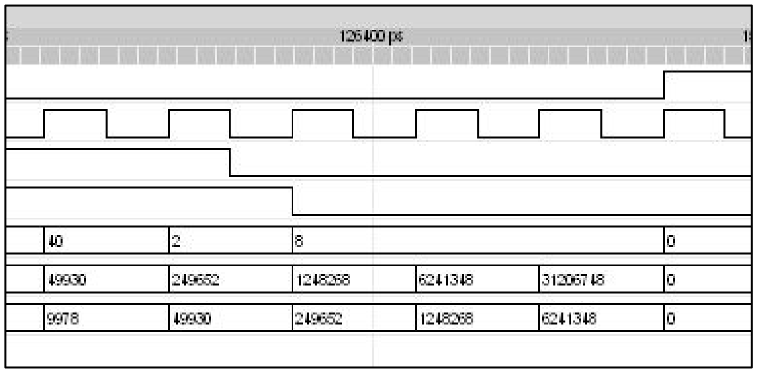


Figure 6.8c. Level 3 (RTL MOC) simulation results for Producer/Consumer model (3/3)

6.1.3 Level 3 to Level 4 Translation

The third and final Perl script constrains the binary bit length of I/O ports and internal variables of the Level 3 SystemC modules and produces a Level 4 model with the uniform bit length of 24. The Level 4 model is synthesis-ready. The three Perl scripts have successfully completed the Ptolemy to SystemC translation.

6.1.4 Post-synthesis Results of Level 5 Model

The SystemC header and source files of the Level 4 model are the inputs to the synthesis tool SystemC Compiler. The synthesized result is a Level 5 model which is a gate-level ASIC circuit. Figures 6.9a through 6.9c show the synthesized simulation results. As can be seen by the results on signal *OUTPUT_INTEGER*, these figures match the benchmark of Figure 6.4.

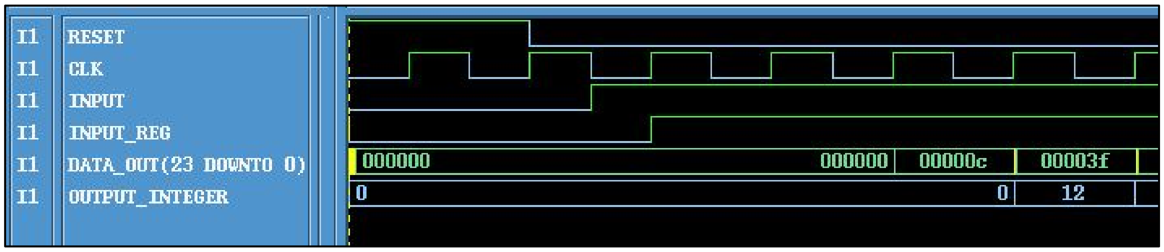


Figure 6.9a. Level 5 (gate-level) simulation results for Producer/Consumer model (1/3)

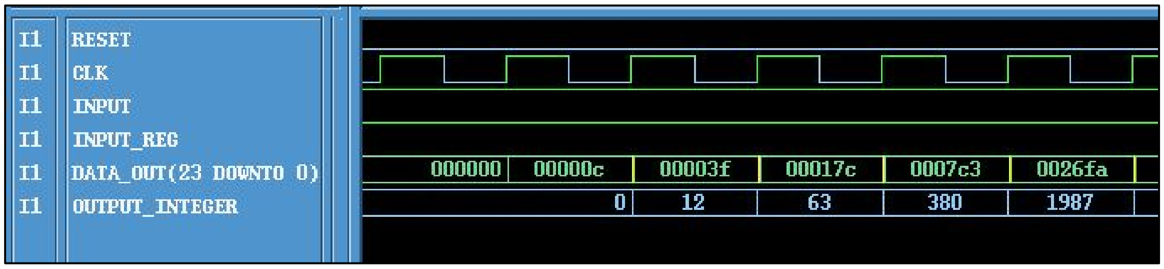


Figure 6.9b. Level 5 (gate-level) simulation results for Producer/Consumer model (2/3)

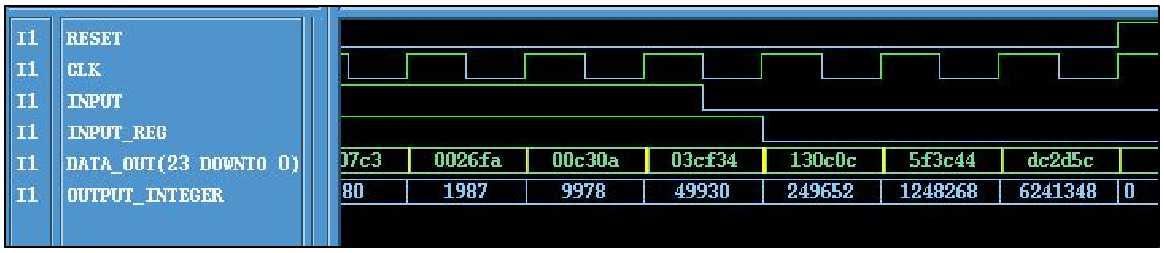


Figure 6.9c. Level 5 (gate-level) simulation results for Producer/Consumer model (3/3)

Because Level 5 is a gate-level model, the overall circuit experiences propagation delays and hazards [34-36]. Propagation delay is generally defined as the time it takes for a signal to travel from one point to another. At the gate level, this is in reference to the time it takes for a signal to propagate from the input to the output of a gate (i.e.: AND gate, OR gate, etc.). During this time period, the output may experience a series of glitches. Since this is an RTL model, these glitches are referred to as essential hazards. These hazards are primarily at the output of a sequential circuit namely register banks. Figure 6.10 illustrates an essential hazard. This is a zoomed in view on the registered signal *OUTPUT_INTEGER* making the clocked transition from 63 to 380 on the rising edge of *CLK*. Note the momentary glitches of 63 to 60 to 380.

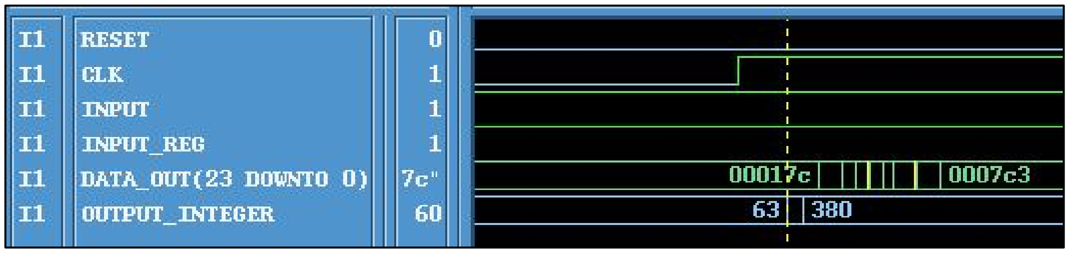


Figure 6.10. Essential hazard of Level 5 (gate-level) model

Figure 6.10 also shows an example of a dynamic hazard caused by combinational logic. Dynamic hazards are basically a series of glitches happening at once (due to individual gate delays) until the output finally settles to the correct state. Signal *DATA_OUT(23 DOWNTO 0)* is the output of the synthesized combinational logic sub-block *AddSubtract* found in the synthesized *SDF_Consumer* block. Along with the gate delays of the *AddSubtract* sub-block, the gate delays of the sub-blocks *MultiplyDivide* (found in the *SDF_Consumer* block) and *ArrayElement* (found in the *DE_Producer* block) contribute to the overall gate delay, and, hence, the series of glitches. Without these glitches, the ideal transition for this signal is $17c_{16}(380_{10})$ to $7c3_{16}(1987_{10})$. Note that these hazards have no adverse effect in a clocked system, because they are eliminated by the rise of the sampling clock which occurs after they settle out.

6.2 HDE-DE-SDF-SR Heterogeneous Ptolemy Model

Figure 6.11 shows the Ptolemy structural model of the second heterogeneous model. This model is composed of 4 domains and their associated directors, 5 composite actors, 7 actors, and 1 modal model (e.g.: a finite state machine). The four domains are HDE, DE, SDF, and Synchronous Reactive (SR) [2,3]. The first three domains were briefly defined in section 6.1. A brief definition of the SR domain is as follows:

SR: This domain is useful for applications with concurrent and complex control logic. It is Synchronous (thus the S) in that it is activated by the clock, but it is Reactive (thus the R) in an instantaneous zero delay manner.

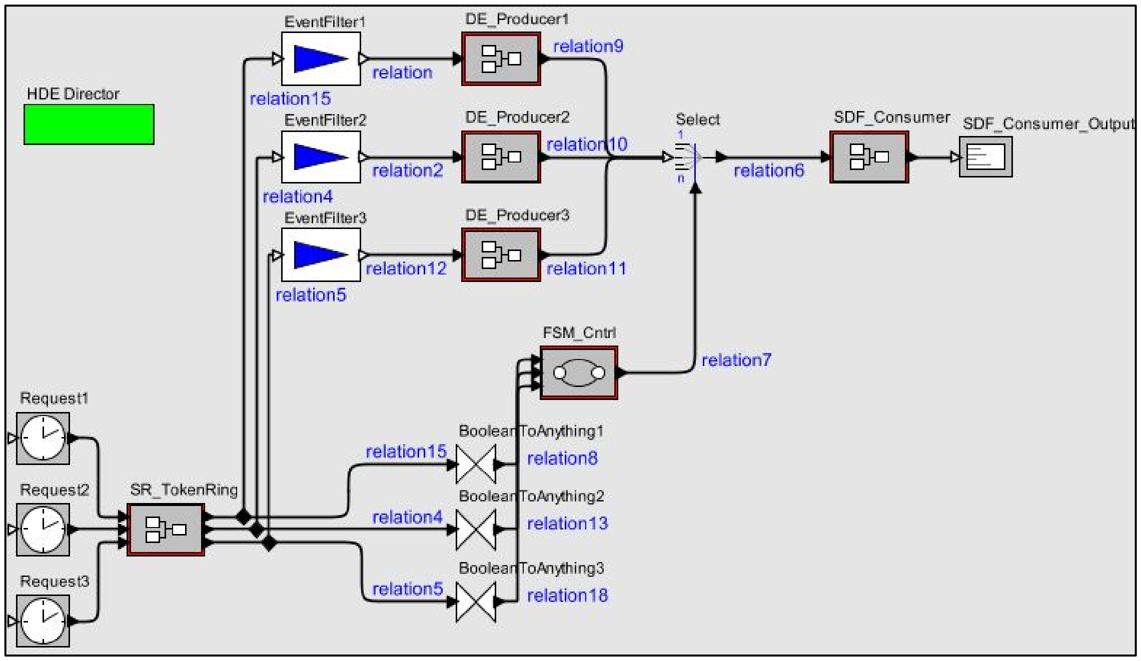


Figure 6.11. HDE-DE-SDF-SR Domain of Producer/Consumer Model

This heterogeneous model basically gives the three producers access to the consumer model. The other actors serve as an elaborate control mechanism to regulate the order in which the producers connect to the consumer. The composite actors *DE_Producer1*, *DE_Producer2*, and *DE_Producer3* represent the three producers while *SDF_Consumer* represents the consumer block. The producers are identical to the one producer in section 6.1 with the exception of the array content in each respective producer. Table 6.1 shows the array elements in the *Data_In* actor of each producer composite actor. The consumer *SDF_Consumer* is identical to the one in section 6.1. This composite actor is a first-order IIR filter with a filter coefficient of 5. The actors *Request1*, *Request2*, and *Request3* are source actors while *SDF_Consumer_Output* is a sink actor. Sections 6.2.1 through 6.2.5 deal with discussing the remaining components which make up the elaborate control mechanism.

Table 6.1. Contents of arrays in DE_Producers

Producer Name	Array Name	Array Contents
DE_Producer1	Data_In1	{0,12,3,65,87,43,40,2,8,28,35}
DE_Producer2	Data_In2	{0,-12,-3,-65,-87,-43,-40,-2,-8,-28,-35}
DE_Producer3	Data_In3	{0,1,1,1,1,1,1,1,1,1,1}

6.2.1 *SR_TokenRing* Composite Actor

This composite actor is under the SR director. It models a basic cyclic token-ting arbiter that arbitrates fairly among requests by marching a token around a ring [37,38]. Specifically, this composite actor indirectly controls the composite actors *DE_Producer1*, *DE_Producer2*, and *DE_Producer3* by dictating when one of these producers is to output an integer to the *Select* actor. Figure 6.12 shows the actor components of this composite actor. The actors inside this composite actor work on the Boolean tokens coming from the *Clock* actors in Figure 6.11 (*Request1*, *Request2*, and *Request3*).

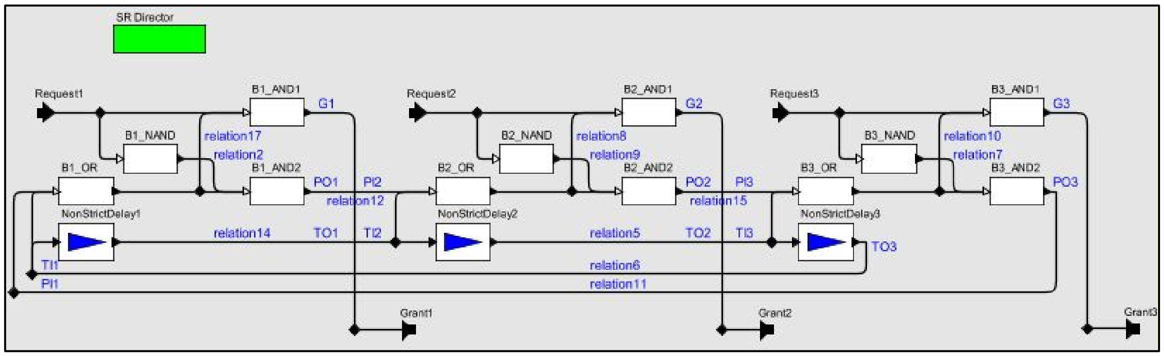


Figure 6.12. Cyclic token-ring arbiter model inside composite actor *SR_TokenRing*

6.2.2 *EventFilter* Actors

Under the HDE director, three *EventFilter* actors are shown in Figure 6.11. These actors have the user-defined names of *EventFilter1*, *EventFilter2*, and *EventFilter3*. Under Ptolemy, these actors operate on Boolean tokens and provide the function of outputting a true token when a true token is received at its input. False tokens are discarded by this actor, and, as a result, no token is produced at the output [2]. These actors are controlled by the tokens coming from the *SR_TokenRing* composite actor. The *EventFilter* actors then determine if true tokens are needed to trigger any of the producer composite actors.

6.2.3 *BooleanToAnything* Actors

Under the HDE director, there are three *BooleanToAnything* actors with the user-defined names of *BooleanToAnything1*, *BooleanToAnything2*, and *BooleanToAnything3*. In this heterogeneous model, each actor takes as input either a true or false token and outputs either an integer 1 token or integer 0 token respectively [2]. These Ptolemy actors produce output tokens that will act as input tokens to the modal model *FSM_Cntrl*.

6.2.4 *FSM_Cntrl* Modal Model

This model is called a *modal model* [2]. In this heterogeneous model, *FSM_Cntrl* represents a Mealy finite state machine (FSM) containing four states. Based on the current state (i.e.: *init*, *s*, *s2*, or *s3*) and the input Boolean tokens from the *BooleanToAnything* actors, the modal model *FSM_Cntrl* outputs an integer between 0 and 2. This integer output then serves as input to the Ptolemy *Select* actor. Figure 6.13 shows the finite state machine of this modal model.

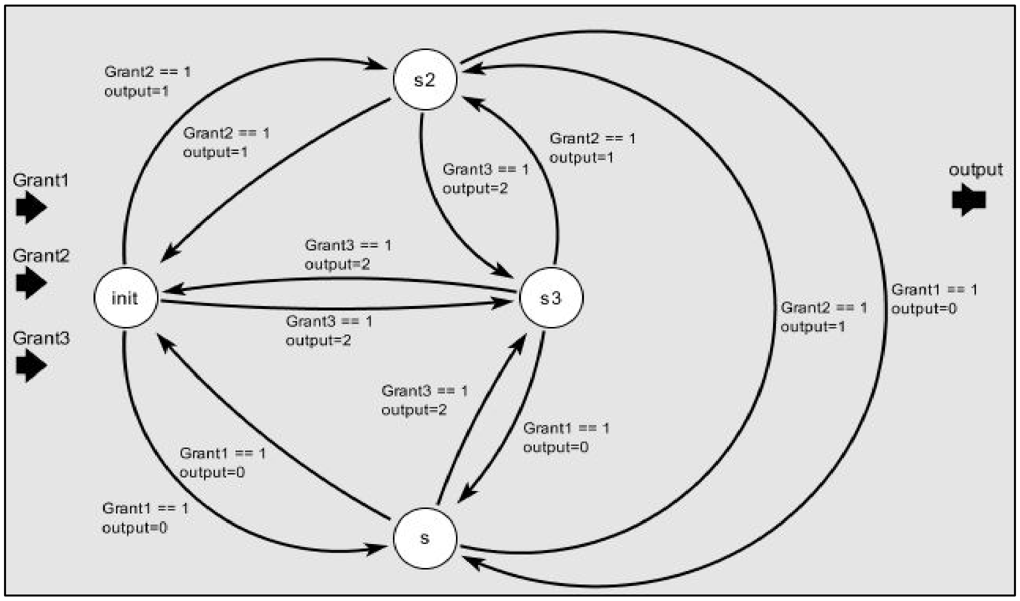


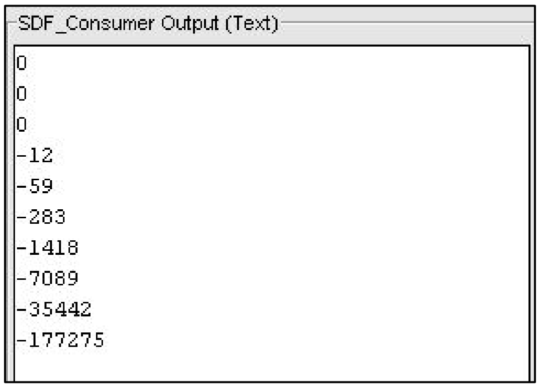
Figure 6.13. FSM contents of *FSM_Cntrl* Modal Model

6.2.5 *Select* Actor

Under HDE director, the *Select* actor serves as a multiplexer [2]. There are three *input* ports and they are driven by the relations *relation9*, *relation10*, and *relation11* as seen in Figure 6.11. These relations connect the *Select* multi-input port to the output ports of the composite actors *DE_Producer1*, *DE_Producer2*, and *DE_Producer3* respectively. The relation *relation7* connects the *control* port of the *Select* actor to the output port of the *FSM_Cntrl* modal model. Based on the integer token value on *relation7*, the integer token value on either *relation9*, *relation10*, or *relation11* is routed to the *output* port of the *Select* actor and subsequently on to *relation6*.

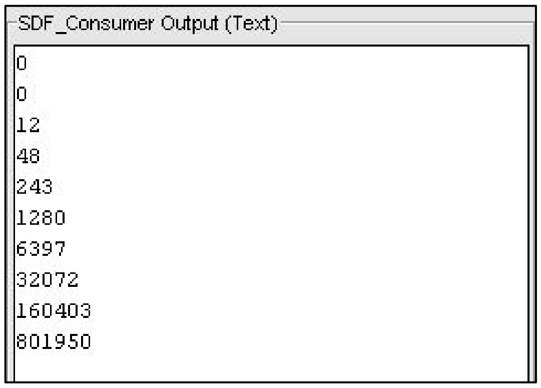
6.2.6 Configuration of Source Actors for enabling Producers

The source actors *Request1*, *Request2*, and *Request3* determine which of the three producers inputs to the consumer. The source actors produce Boolean tokens. Setting the source actor *Request1* to true enables *DE_Producer1* to have access to the consumer while setting it to false disables *DE_Producer1*. Source actors *Request2* and *Request3* enable composite actors *DE_Producer2* and *DE_Producer3*, respectively. Figure 6.14 through 6.16 show three different producer-to-consumer configurations. Figure 6.14 deals with enabling all three producers; Figure 6.15 deals with enabling *DE_Producer1* and *DE_Producer2*; and Figure 6.16 deals with enabling *DE_Producer1*. Note this last configuration is equivalent to the heterogeneous Ptolemy model in section 6.1 in terms of output response when comparing Figure 6.16 with Figure 6.4.


SDF_Consumer Output (Text)

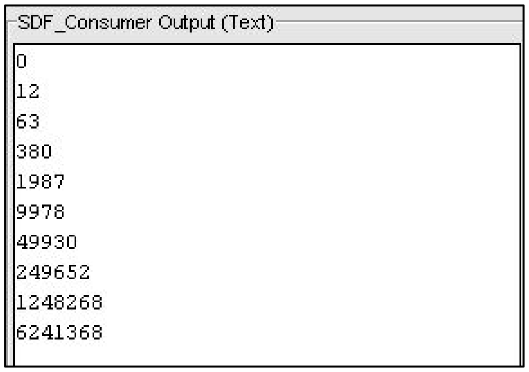
```
0
0
0
-12
-59
-283
-1418
-7089
-35442
-177275
```

Figure 6.14. *Request1*=true; *Request2*=true; *Request3*=true


SDF_Consumer Output (Text)

```
0
0
12
48
243
1280
6397
32072
160403
801950
```

Figure 6.15. *Request1*=true; *Request2*=true; *Request3*=false

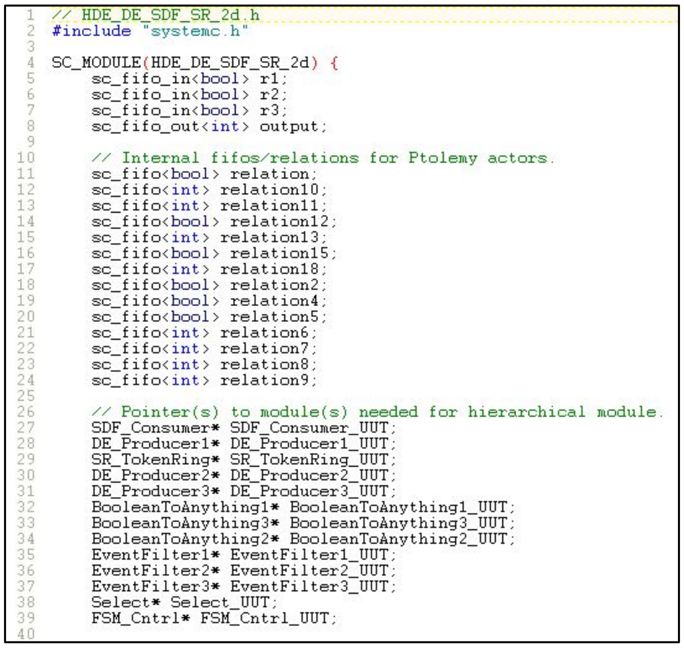


```
SDF_Consumer Output (Text)
0
12
63
380
1987
9978
49930
249652
1248268
6241368
```

Figure 6.16. *Request1*=true; *Request2*= false; *Request3*=false

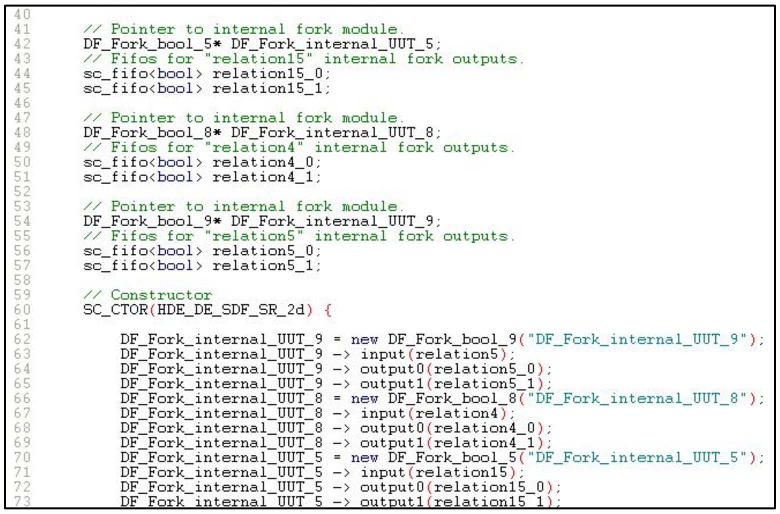
6.2.7 Level 1 to Level 2 Translation

The first Perl script performs the initial level-to-level translation. Figures 6.17a through 6.17d show the SystemC hierarchical module of Figure 6.11. Figure 6.18 shows the simulation results for this model under the KPN model of computation when all three producers are enabled. As can be seen by the results, the simulation identically matches the benchmark of Figure 6.14.



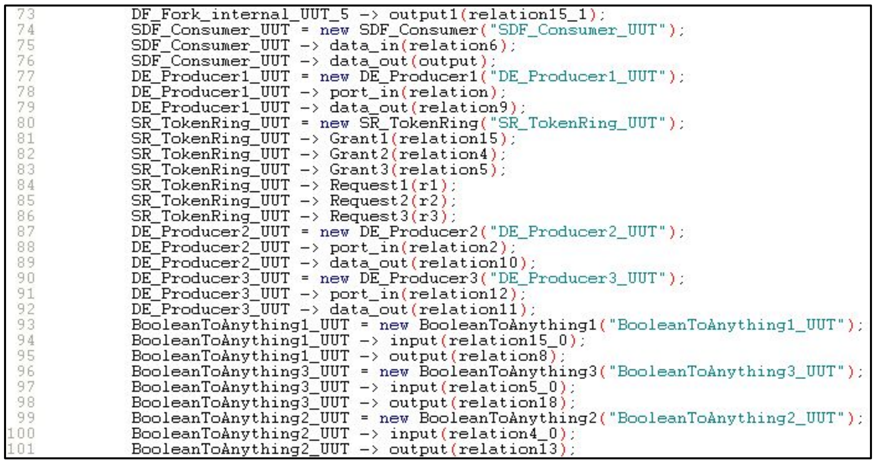
```
// HDE_DE_SDF_SR_2d.h
#include "systemc.h"

SC_MODULE(HDE_DE_SDF_SR_2d) {
    sc_fifo_in<bool> r1;
    sc_fifo_in<bool> r2;
    sc_fifo_in<bool> r3;
    sc_fifo_out<int> output;

    // Internal fifos/relations for Ptolemy actors.
    sc_fifo<bool> relation;
    sc_fifo<int> relation10;
    sc_fifo<int> relation11;
    sc_fifo<bool> relation12;
    sc_fifo<int> relation13;
    sc_fifo<bool> relation15;
    sc_fifo<int> relation18;
    sc_fifo<bool> relation2;
    sc_fifo<bool> relation4;
    sc_fifo<bool> relation5;
    sc_fifo<int> relation6;
    sc_fifo<int> relation7;
    sc_fifo<int> relation8;
    sc_fifo<int> relation9;

    // Pointer(s) to module(s) needed for hierarchical module.
    SDF_Consumer* SDF_Consumer_UUT;
    DE_Producer1* DE_Producer1_UUT;
    SR_TokenRing* SR_TokenRing_UUT;
    DE_Producer2* DE_Producer2_UUT;
    DE_Producer3* DE_Producer3_UUT;
    BooleanToAnything1* BooleanToAnything1_UUT;
    BooleanToAnything3* BooleanToAnything3_UUT;
    BooleanToAnything2* BooleanToAnything2_UUT;
    EventFilter1* EventFilter1_UUT;
    EventFilter2* EventFilter2_UUT;
    EventFilter3* EventFilter3_UUT;
    Select* Select_UUT;
    FSM_Cntrl* FSM_Cntrl_UUT;

```

Figure 6.17a. SystemC hierarchical module of Producer/Consumer model (1/4)



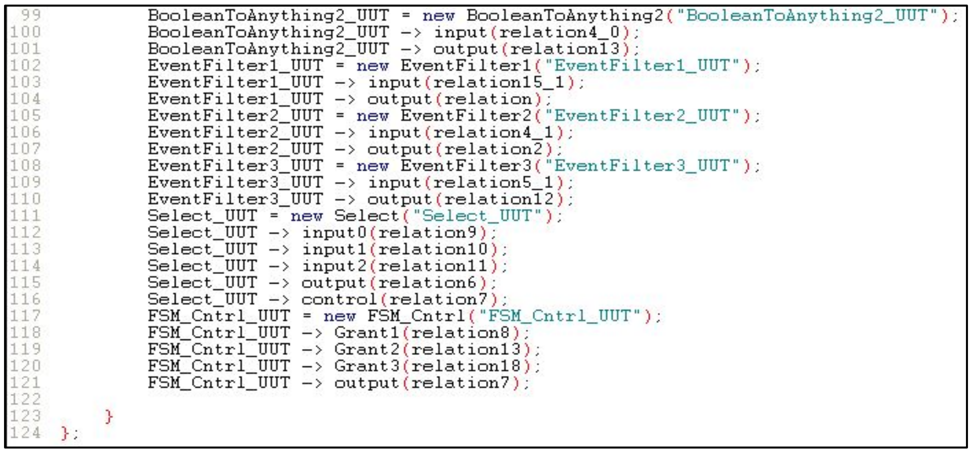
```

    // Pointer to internal fork module.
    DF_Fork_bool_5* DF_Fork_internal_UUT_5;
    // Fifos for "relation15" internal fork outputs.
    sc_fifo<bool> relation15_0;
    sc_fifo<bool> relation15_1;

    // Pointer to internal fork module.
    DF_Fork_bool_8* DF_Fork_internal_UUT_8;
    // Fifos for "relation4" internal fork outputs.
    sc_fifo<bool> relation4_0;
    sc_fifo<bool> relation4_1;

    // Pointer to internal fork module.
    DF_Fork_bool_9* DF_Fork_internal_UUT_9;
    // Fifos for "relation5" internal fork outputs.
    sc_fifo<bool> relation5_0;
    sc_fifo<bool> relation5_1;

    // Constructor
    SC_CTOR(HDE_DE_SDF_SR_2d) {

        DF_Fork_internal_UUT_9 = new DF_Fork_bool_9("DF_Fork_internal_UUT_9");
        DF_Fork_internal_UUT_9 -> input(relation5);
        DF_Fork_internal_UUT_9 -> output0(relation5_0);
        DF_Fork_internal_UUT_9 -> output1(relation5_1);
        DF_Fork_internal_UUT_8 = new DF_Fork_bool_8("DF_Fork_internal_UUT_8");
        DF_Fork_internal_UUT_8 -> input(relation4);
        DF_Fork_internal_UUT_8 -> output0(relation4_0);
        DF_Fork_internal_UUT_8 -> output1(relation4_1);
        DF_Fork_internal_UUT_5 = new DF_Fork_bool_5("DF_Fork_internal_UUT_5");
        DF_Fork_internal_UUT_5 -> input(relation15);
        DF_Fork_internal_UUT_5 -> output0(relation15_0);
        DF_Fork_internal_UUT_5 -> output1(relation15_1);
```

Figure 6.17b. SystemC hierarchical module of Producer/Consumer model (2/4)



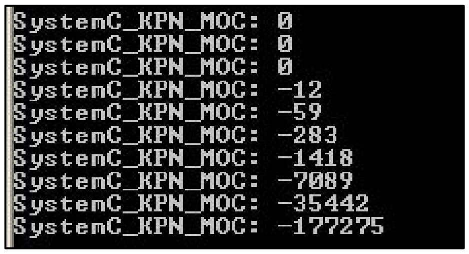
```
        DF_Fork_internal_UUT_5 -> output1(relation15_1);
        SDF_Consumer_UUT = new SDF_Consumer("SDF_Consumer_UUT");
        SDF_Consumer_UUT -> data_in(relation6);
        SDF_Consumer_UUT -> data_out(output);
        DE_Producer1_UUT = new DE_Producer1("DE_Producer1_UUT");
        DE_Producer1_UUT -> port_in(relation);
        DE_Producer1_UUT -> data_out(relation9);
        SR_TokenRing_UUT = new SR_TokenRing("SR_TokenRing_UUT");
        SR_TokenRing_UUT -> Grant1(relation15);
        SR_TokenRing_UUT -> Grant2(relation4);
        SR_TokenRing_UUT -> Grant3(relation5);
        SR_TokenRing_UUT -> Request1(r1);
        SR_TokenRing_UUT -> Request2(r2);
        SR_TokenRing_UUT -> Request3(r3);
        DE_Producer2_UUT = new DE_Producer2("DE_Producer2_UUT");
        DE_Producer2_UUT -> port_in(relation2);
        DE_Producer2_UUT -> data_out(relation10);
        DE_Producer3_UUT = new DE_Producer3("DE_Producer3_UUT");
        DE_Producer3_UUT -> port_in(relation12);
        DE_Producer3_UUT -> data_out(relation11);
        BooleanToAnything1_UUT = new BooleanToAnything1("BooleanToAnything1_UUT");
        BooleanToAnything1_UUT -> input(relation15_0);
        BooleanToAnything1_UUT -> output(relation8);
        BooleanToAnything3_UUT = new BooleanToAnything3("BooleanToAnything3_UUT");
        BooleanToAnything3_UUT -> input(relation5_0);
        BooleanToAnything3_UUT -> output(relation18);
        BooleanToAnything2_UUT = new BooleanToAnything2("BooleanToAnything2_UUT");
        BooleanToAnything2_UUT -> input(relation4_0);
        BooleanToAnything2_UUT -> output(relation13);
```

Figure 6.17c. SystemC hierarchical module of Producer/Consumer model (3/4)



```
        BooleanToAnything2_UUT = new BooleanToAnything2("BooleanToAnything2_UUT");
        BooleanToAnything2_UUT -> input(relation4_0);
        BooleanToAnything2_UUT -> output(relation13);
        EventFilter1_UUT = new EventFilter1("EventFilter1_UUT");
        EventFilter1_UUT -> input(relation15_1);
        EventFilter1_UUT -> output(relation);
        EventFilter2_UUT = new EventFilter2("EventFilter2_UUT");
        EventFilter2_UUT -> input(relation4_1);
        EventFilter2_UUT -> output(relation2);
        EventFilter3_UUT = new EventFilter3("EventFilter3_UUT");
        EventFilter3_UUT -> input(relation5_1);
        EventFilter3_UUT -> output(relation12);
        Select_UUT = new Select("Select_UUT");
        Select_UUT -> input0(relation9);
        Select_UUT -> input1(relation10);
        Select_UUT -> input2(relation11);
        Select_UUT -> output(relation6);
        Select_UUT -> control(relation7);
        FSM_Cntrl_UUT = new FSM_Cntrl("FSM_Cntrl_UUT");
        FSM_Cntrl_UUT -> Grant1(relation8);
        FSM_Cntrl_UUT -> Grant2(relation13);
        FSM_Cntrl_UUT -> Grant3(relation18);
        FSM_Cntrl_UUT -> output(relation7);

    }
};
```

Figure 6.17d. SystemC hierarchical module of Producer/Consumer model (4/4)



```
SystemC_KPN_MOC: 0
SystemC_KPN_MOC: 0
SystemC_KPN_MOC: 0
SystemC_KPN_MOC: -12
SystemC_KPN_MOC: -59
SystemC_KPN_MOC: -283
SystemC_KPN_MOC: -1418
SystemC_KPN_MOC: -7089
SystemC_KPN_MOC: -35442
SystemC_KPN_MOC: -177275
```

Figure 6.18. Level 2 (KPN MOC) simulation results for Producer/Consumer model

6.2.8 Level 2 to Level 3 Translation

This second Perl script takes as input the initial SystemC source and header files of the Level 2 model and introduces synthesizable SystemC constructs. The result is a Level 3 representation. Figures 6.19a and 6.19b show the simulation results for this model under the RTL model of computation when all three producers are enabled. As can be seen by the results on the output signal *SystemC.output[31:0]*, the simulation identically matches the benchmark of Figure 6.14.

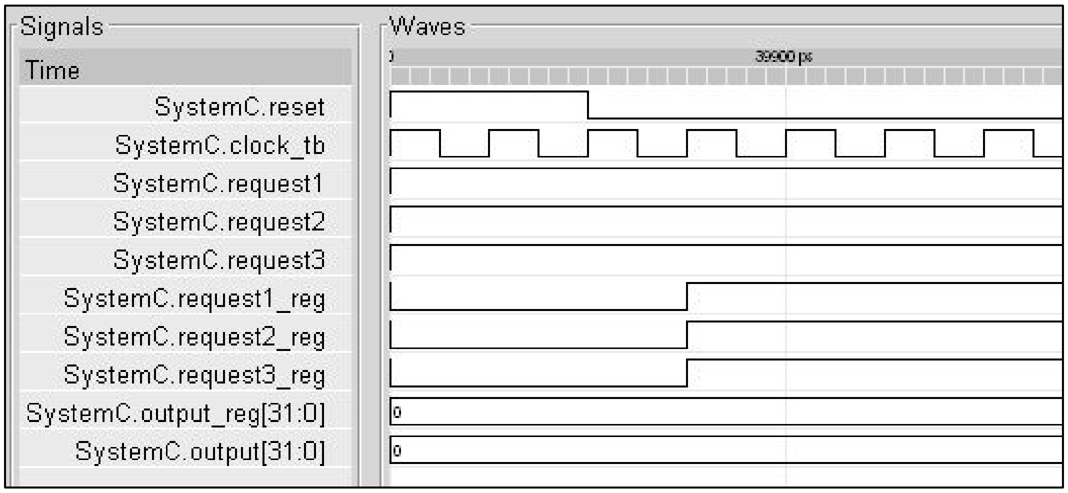


Figure 6.19a. Level 3 (RTL MOC) simulation results for Producer/Consumer model (1/2)

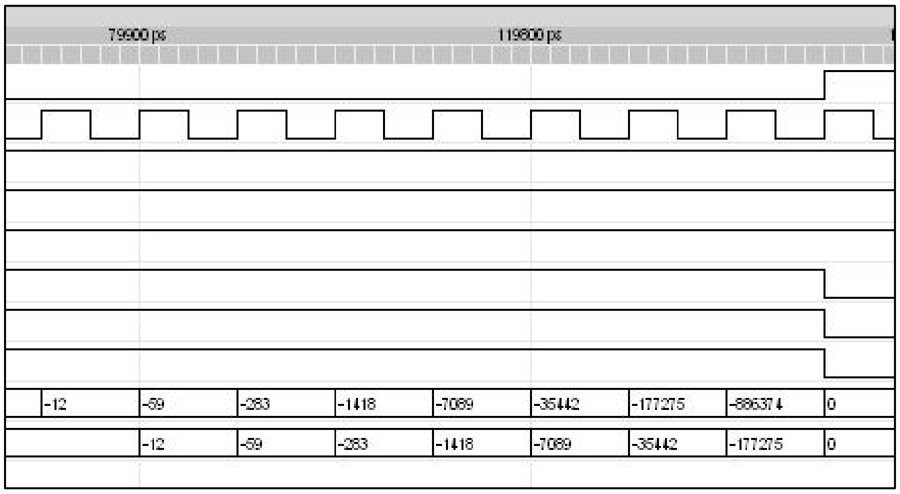


Figure 6.19b. Level 3 (RTL MOC) simulation results for Producer/Consumer model (2/2)

6.2.9 Level 3 to Level 4 Translation

The third and final Perl script constrains the binary bit length of I/O ports and internal variables of the Level 3 SystemC modules and produces a Level 4 model with the uniform bit length of 26. The Level 4 model is synthesis-ready. The three Perl scripts have successfully completed the Ptolemy to SystemC translation.

6.2.10 Post-synthesis Results of Level 5 Model

The SystemC header and source files of the Level 4 model are the inputs to the synthesis tool SystemC Compiler. Figures 6.20a through 6.20c show the synthesized simulation results when all three producers are enabled. As can be seen by the results on signal *OUTPUT_INTEGER*, these figures match the benchmark of Figure 6.14 along with glitches.

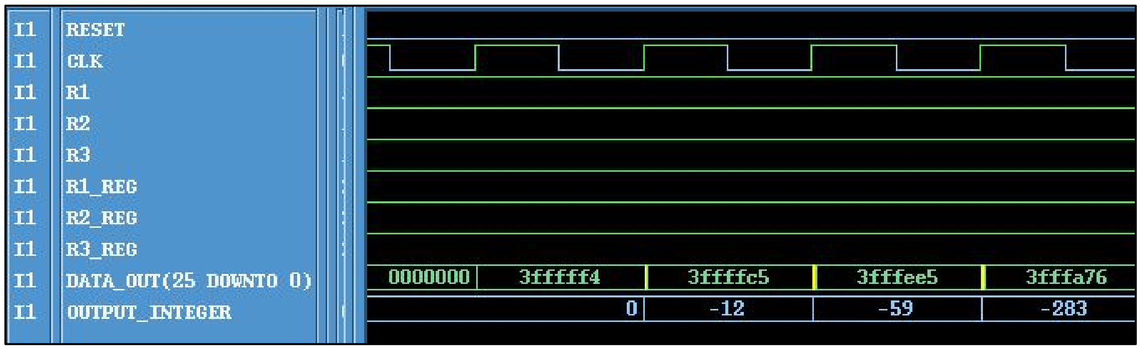


Figure 6.20a. Level 5 (gate-level) simulation results for Producer/Consumer model (1/3)

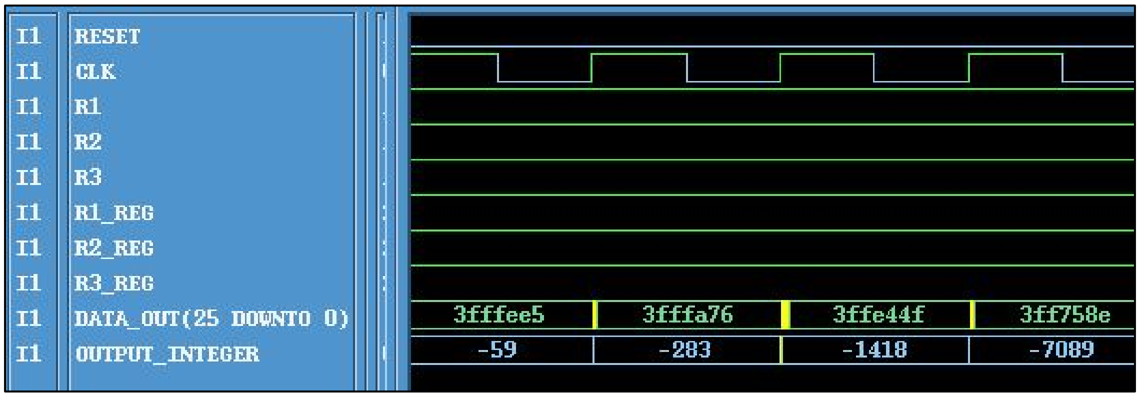


Figure 6.20b. Level 5 (gate-level) simulation results for Producer/Consumer model (2/3)

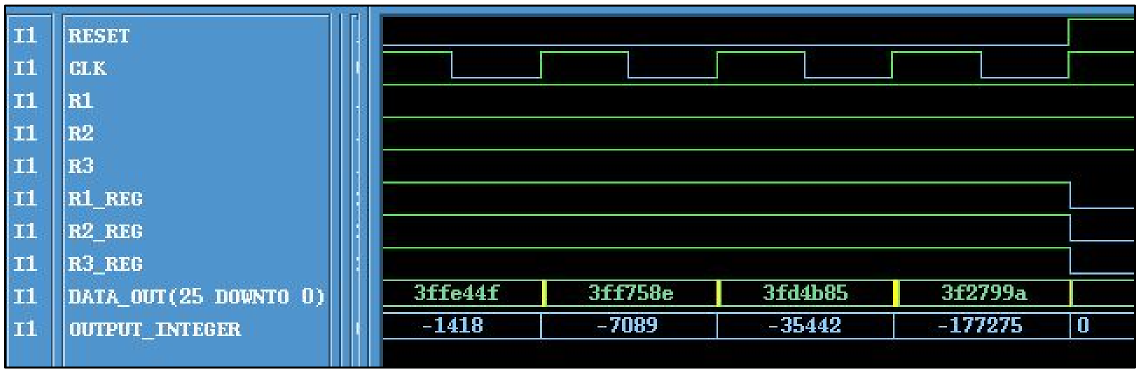


Figure 6.20c. Level 5 (gate-level) simulation results for Producer/Consumer model (3/3)

Chapter 7. Summary and Conclusions and Recommendations for Future Work

Developing a transformation methodology using XML and Perl as tools to translate Ptolemy models into synthesized SystemC code for input to the SystemC Compiler (which synthesizes the code into a gate-level circuit) was the objective of this research. The goal of moving a high-level of abstraction (Ptolemy model) to a low-level of abstraction (synthesizable SystemC code) was achieved. The defined 5-level design process provides a technical bridge between top-down and bottom-up design methodologies. This technical bridge provided by the transformation methodology capitalized on the major advantages of the two design methodologies. Those major advantages are fast design exploration, efficient IP-reuse, and validation.

This chapter summarizes the results and transformation methodology of this research. In addition, conclusions are drawn as well as suggestions for future work to further improve the transformation methodology.

7.1 Summary of Results and Transformation Methodology

The results of this research yielded working gate-level models that would fit well in the ASIC/FPGA design industry. Specifically, in this research, these gate-level models are synthesized ASIC designs. The transformation methodology obtained by the 5-level design process used a specific high-level modeling tool and a specific system-level design language. The modeling tool and design language are Ptolemy II and SystemC, respectively. As a result, the XML documents, module templates, and Perl scripts are specific and are developed accordingly.

Once a Ptolemy model is considered feasible in hardware, its simulation results are the basis for evaluation, or benchmark, when comparing the simulation results of the lower SystemC models. As was seen by the results, the KPN and RTL models of computation for the SystemC models produce simulation results that identically match the benchmark established by the initial Ptolemy model.

When simulating the gate-level (Level 5) models, the simulation results matched the benchmark of the Ptolemy simulation results. As expected when using ASIC target libraries, time delays and glitches brought on by hazards manifested themselves. But, nonetheless, the gate-level models matched the high-level benchmarks at clock edges, which is the desired behavior. The two heterogeneous Ptolemy models were successfully translated and their behaviors were preserved.

7.2 Conclusions

Understanding the syntax and semantics of a Ptolemy actor under the influence of a Ptolemy director is crucial when translating to the lower levels. Specifically, when proceeding down the 5-level design hierarchy, deciding if a Ptolemy structural model is feasible in hardware is important depending on the director. For certain Ptolemy actors in this research, Perl scripts insert control logic into translated SystemC modules to preserve the behavior of the Ptolemy actor to be translated. The *SampleDelay* actor is a prime example. Preservation of an actor's behavior requires knowledge of the object-oriented language Java. To supplement this, when translating to SystemC, a designer must also have sufficient knowledge of the syntax and semantics of C++ which is another object-oriented language. Understanding both these programming languages is critical when using Perl to perform the actual translations and guarantees that the final SystemC modules (Level 4) are synthesis-ready.

Translating the module template of a Ptolemy actor into a Level 2 SystemC module requires the token dataflow mechanism as the means of communication. Using the SystemC blocking *read()* and *write()* methods support this required dataflow mechanism. These methods implicitly provide the control of token dataflow observed in the Ptolemy counterpart where the director explicitly regulates the flow of tokens, regardless of data type, in a Ptolemy structural model.

When translating down to the RTL levels (as represented by Levels 3 and 4) of the 5-level design process, the use of signal dataflow as the means of model communication introduced a significant pre-synthesis issue. This issue dealt with having a designer, depending on the SystemC module, to explicitly define an initial state/value. Typically, when executing the simulation of a Ptolemy model, the start and end of the simulation represents normal operation. Because of Ptolemy's high-level of abstraction, there is no need to explicitly make sure that all actors in a Ptolemy model each will produce the correct initial state/value. The only exception to this is for those actors that have value-configurable parameters. These parameters explicitly define the initial value of an actor. In RTL modeling, the solution to this pre-synthesis issue is to incorporate the reset mechanism devised in this research. In ASIC/FPGA design, a designer must define the initial conditions of sequential components through a reset mechanism. This reset mechanism serves the purpose of initializing all internal and output signals in an RTL model to a known state/value, and it is at the known state/value condition that an RTL model could be considered to operate normally.

The RTL modeling style introduced *clock latency*. Specifically, since I/O ports of the Levels 3 through 5 models are registered through register banks (i.e.: D-type flip-flops with asynchronous, active-high reset), both inputs and outputs each have a clock latency of one by default. But, depending on the circuit behavior between the I/O register banks, the clock latency for the output ports may be more than one. This is especially true if the circuit contains some sequential logic.

Overall, the transformation methodology created in this research did successfully translate the high-level of abstraction represented by Ptolemy models to the low-level of abstraction represented by synthesizable SystemC code. The synthesized gate-level model did further validate the transformation methodology (through simulation) by showing it is possible to create a comprehensive, robust methodology and environment to combine both top-down and bottom-up design methodologies. The high-level models represented by Ptolemy models were able to be mapped to lower level core models. These core models came from the gate-level models found in the ASIC target library used in this research. The actual mapping at this gate-level was performed by the synthesis tool SystemC Compiler. With the specific use of the modeling tool Ptolemy II and the system-level design language SystemC, it is generally possible to combine the specification-oriented approach of top-down design with the core-reuse based approach of bottom-up design. Specifically, in this research, the core-reuse based approach is in regards to two domains. The first domain deals with the first Perl script which uses module templates while the second domain deals with target libraries at the synthesis level. For the first (domain) Perl script, the module templates serve as a design library from which the initial Ptolemy to SystemC translation is made.

7.3 Suggestions for Future Work

Based on the results of this research, future works are suggested to improve the transformation methodology. These suggestions are specific to the use of Ptolemy and SystemC together.

7.3.1 Increased Productivity of Ptolemy to SystemC Translation

For future versions of Ptolemy, it would be advantageous to incorporate those parts of the transformation methodology that involve Perl scripting into the overall Ptolemy software package. The major advantage would be speed in terms of quickly producing synthesizable SystemC code after a Ptolemy model is deemed hardware-feasible and its behavior meets the required design specification.

7.3.2 Test Bench Generation for Level 2

The first Perl script used in this thesis research ignores the source and sink actors in a Ptolemy structural model. In the future research, source actors would not be ignored in this initial translation from Ptolemy to SystemC and, hence, be translated into SystemC source modules. These modules would be testbench components used to provide input stimuli (tokens) into the Level 2 model.

7.3.3 Test Bench Generation for Levels 3 and 4

A new Perl script would be created that automatically transforms the testbench source modules from Level 2 into RTL modules for Levels 3 and 4 (i.e.: incorporate resets and clocking).

References

[1] Virtual Socket Interface Alliance (VSIA): http://www.vsi.org.

[2] E. A. Lee, et al., “Ptolemy II, Heterogeneous Current Modeling and Design in JAVA,” *Technical Memorandum UCB/ERL M04/27, July 29, 2004.*

[3] E. A. Lee and S. Neuendorffer, “Concurrent Models of Computation for Embedded Software," *Technical Memorandum UCB/ERL M04/26, July 22, 2004.*

[4] T. Grotker, S. Liao, G. Martin, S. Swan, *System Design with SystemC.* Norwell, MA: Kluwer 2002.

[5] SystemC Consortium: http://www.systemc.org.

[6] G. Kahn, “The semantics of a simple language for parallel programming," In J. L. Rosenfeld, editor, *Information Processing*. North-Holland Publishing Company, 1974.

[7] J. T. Buck, *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. Ph.d. thesis, University of California, Berkeley, 1973.

[8] Synopsys, Incorporated: http://www.synopsys.com.

[9] *SystemC 2.0.1 Language Reference Manual*. Retrieved from http://www.systemc.org/.

[10] J. R. Armstrong and F. G. Gray, *VHDL Design Representation and Synthesis*. Englewood Cliffs, N.J., Prentice Hall, 2000.

[11] D. Gijski, *High Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers; 1st edition (April 2002).

[12] J. K. Aggarwal, *Digital Signal Processing*. Western Periodicals Company; 1979.

[13] D. G. Manolakis, J. G. Proakis, “Digital Signal Processing: Principles, Algorithms, and Applications”, Macmillan Publishing Company, 1992.

[14] J. R. Armstrong and U. Ronen, "Modeling With SystemC: A Case Study,” Proceeding of HDL 2001, San Jose, CA, pp. 158-164.

[15] E. Castro, *XML for the World Wide Web: Visual QuickStart Guide*. Berkeley, CA: Peachpit Press.

[16] The World Wide Web Consortium (W3C): http://www.w3.org.

[17] C. Von See and N. Keskar, *XSLT Developer’s Guide*. Berkeley, CA: McGraw-Hill/Osborne

[18] P. H. Winston, *On To C++*. Reading, MA: Addison-Wesley Publishing Company.

[19] J. Smiley, *Learn to Program with Java*. New York, NY: McGraw-Hill/Osborne.

[20] P. H. Winston and S. Narasimhan, *On To Java*. Boston, MA: Addison-Wesley Publishing Company.

[21] L. Lemay, *Sams Teach Yourself Perl in 21 Days*. Indianapolis, IN: Sams Publishing 2002.

[22] N. Chapman, *PERL, The Programmer’s Companion*. New York, NY: John Wiley & Sons, Inc. 1997.

[23] R. F. Tinder, *Engineering Digital Design*. San Diego, CA: Academic Press, 2000.

[24] H. Lam and J. O’Malley, *Fundamentals of Computer Engineering*. John Wiley & Sons, Inc. 1988.

[25] *SystemC 1.1 User’s Guide*. Retrieved from http://www.systemc.org/.

[26] Altova XMLSpy: http://www.altova.com.

[27] D. D. Gajski, et. al., *Specification and Design of Embedded Systems*. PTR Prentice Hall, Inc. 1994.

[28] S. Y. Liao, "Towards a New Standard for System-Level Design," Proceedings of the 8th International Workshop on Hardware/Software Codesign, San Diego, CA, pp. 2-6.

[29] R. J. Wilson, *Introduction to Graph Theory,* Academic Press, NY, 1972.

[30] D. J. West, *Introduction to Graph Theory,* Prentice-Hall, Inc.; 2nd edition, 2001.

[31] A. Lee and A. Sangiovanni-Vincentelli, "A Framework for Comparing Models of Computation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 17. No 12. December 1998, pp 1217-1229.

[32] C. Akturan and M. F. Jacome, "RS-FDRA: A Register-Sensitive Software Pipelining Algorithm for Embedded VLIW Processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 21. No 12. December 2002, pp 1399-1400.

[33] J. R. Armstrong, "Denotational and Operational Semantics of Process Model Graphs," White Paper, copy at http://www.visc.vt.edu/armstrong/index.html.

[34] D. D. Givone, *Digital Principles and Design*. The McGraw-Hill Companies, Inc. 2003.

[35] D. Winkel and F. Prosser, *The Art of Digital Design*. Prentice-Hall, Inc. 1980.

[36] A. Miczo, *Digital Logic Testing and Simulation*. John Wiley & Sons, Inc.; 2nd edition, 2003.

[37] J. T. Carlo, R. D. Love, "Understanding Token Ring Protocols and Standards", Artech House Publishers, Inc., 1998.

[38] P. Whitaker, "The Simulation of Synchronous Reactive Systems in Ptolemy II," Master's Report, Memorandum UCB/ERL M01/20, Electronics Research Laboratory, University of

California, Berkeley, May 2001. (http://ptolemy.eecs.berkeley.edu/publications/papers/01/sr/).

[39] Y. Taur, “The Incredible Shrinking Transistor,” IEEE Spectrum, Vol. 36, Issue 7, July 1999, pp. 25–29.

[40] M. Lapedus, "Intel develops 20-nm transistor 20GHz processor by 2007", SemiConductor Business News, June 14, 2001.

[41] GTKWave Analyzer: http://www.cs.manchester.ac.uk/apt/.

[42] *CoCentric SystemC Compiler*. Retrieved from http://www.sdrforum.org/MTGS/wde_wkshp_11_00/dac00_cocentric_systemc_compiler_12_1_00.pdf.

[43] G. Martin and B. Salefski, “System Level Design for SOC’s: A Progress Report, Two Years On," *International HDL Conference and Exhibition*, 2000, pp. 3-10.

[44] W. Mueller, et. al., “The Simulation Semantics of SystemC," *Proceedings Design Automation and Test in Europe*, 2001, pp. 64-70.

[45] Synopsys, Inc., *CoCentric SystemC Compiler, Behavioral User Guide*, March 2001.

[46] J. Lui and E. A. Lee, “Component-based Hierarchical Modeling of Systems with Continuous and Discrete Dynamics," *Proc. of the 2000 IEEE International Conference on Control Applications and IEEE Symposium on Computer-Aided Control System Design* (CCA/CACSD’00), Anchorage, AK, September 25-27, 2000. pp 95-100.

[47] L. Muliadi, “Discrete Event Modeling in Ptolemy II," MS Report, Dept. of EECS, University of California, Berkeley, CA 94720, May 1999. (http://ptolemy.eecs.berkeley.edu/publications/papers/99/deModeling/).

[48] E. A. Lee and Y. Xiong, “System-Level Types for Component-Based Design," *First Workshop on Embedded Software*, EMSOFT 2001, Lake Tahoe, CA, USA, Oct. 8-10,

2001. (also Technical Memorandum UCB/ERL M00/8, Electronics Research Lab, University of California, Berkeley, CA 94720, February 29, 2000. (http://ptolemy.eecs.berkeley.edu/publications/papers/01/systemLevelType/).

[49] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," *Proc. of the IEEE*, September 1987.

[50] E. A. Lee and T. M. Parks, "DataFlow Process Networks," *Proc. of the IEEE*, 83:773-801, 1987.

[51] D. Ramanathan, R. Roth, and R. Gupta, "Interfacing Hardware and Software using C++ Class Libraries," *Proceedings of the International Conference on Computer Design*, 2000, pp. 445-450.

[52] S. Edwards, et. al., "Design of Embedded Systems: Formal Models, Validation and Synthesis," *Proceedings of the IEEE*, 85(3):366-390, March 1997.