RICE UNIVERSITY

Assertion-Based Flow Monitoring of SystemC Models

by

Sonali Dutta

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Moshe Y. Vardi, Chair
Karen Ostrum George Distinguished
Service Professor in Computational
Engineering, Rice University

Swarat Chaudhuri
Assistant Professor of Computer Science,
Rice University

Luay K. Nakhleh
Associate Professor of Computer Science
and Biochemistry and Cell Biology, Rice
University

Houston, Texas

September, 2013

ABSTRACT

Assertion-Based Flow Monitoring of SystemC Models

by

Sonali Dutta

SystemC is the de facto system modeling language, used to model hardware-software systems. Verification of SystemC models enables early verification of hardware/software interfaces. Assertion-Based Dynamic Verification (ABDV) is a technique that allows dynamic verification of formal properties about the system by automatically generating runtime monitors from those properties.

A concurrent and reactive hardware-software system performs different "jobs" during its execution. Each such job begins with a set of input data, flows through different processes in the system, and finally produces a set of output data. We call such a job a *flow*, since it flows from one SystemC process to another. Flows are dynamic and concurrent, since a flow can begin anytime during the simulation and the system can process multiple flows at the same time.

We provide a library, using which one can explicitly implement flows in a SystemC model or annotate flows in an existing SystemC model with minimal modification. We also provide an automated monitoring framework that let us monitor properties of flows. Such properties capture the reactive nature of a system naturally and are intuitive to write. Our experimental results show that our framework puts minimal runtime overhead of monitoring.

Acknowledgements

I owe my deepest gratitude to my advisor Prof. Moshe Vardi for guiding me through this wonderful journey of my masters degree. Without his kind support and continuous encouragement, this dissertation would have not been possible. I am thankful for his kindness and support through good and bad times and I feel fortunate to have him as my advisor.

I would like to express my greatest appreciation to my thesis committee members Prof. Luay Nakhleh and Prof. Swarat Chaudhuri for their helpful comments and suggestions to improve my work. I am also grateful to Prof. Ron Goldman for his help and guidance. I would like to offer my special thanks to Dr. Deian Tabakov for explaining his work to me and helping me understand it.

I would like to thank my colleagues in the Rice CAVR group Sumit Nain, Kuldeep Meel, Giuseppe Perelli, Dror Fried, Jianwen Li, Morteza Lahijanian for being such good friends and so much fun to work with. I owe my special thanks to our staff member Arnetta Jones for her support.

Finally I would like to express my respect and love to my parents Somnath Dutta and Sipra Dutta, my sister Priyanka Dutta, my grandmother Shefali Dutta and my late grandfather Madhav Lal Sil for always loving me and giving me the strength to achieve the goals of my life.

Contents

Illustrations

Tables

Chapter 1

Introduction

1.1 Background and Motivation

1.1.1 SystemC

SystemC (IEEE Standard 1666-2005) has emerged as the de facto standard for modeling hardware-software systems [1]. SystemC is implemented as a C++ library, which defines macros and base classes for modeling different hardware elements, like modules, processes, channels, signals, ports etc. As a result, SystemC allows users to model both hardware and software components of a system using a single modeling language. This enables easier and early verification of hardware-software interfaces. Using SystemC, one can start designing the software early in the design cycle without waiting for the hardware to get fabricated, enabling hardware-software co-design. Also SystemC makes it possible to co-simulate the hardware and software together very early in the design cycle by building a SystemC prototype of the hardware. SystemC also allows the user to model hardware at different abstraction levels such as transaction level and register-transfer level [2]. SystemC is used in industry to build hardware prototypes for firmware[3].

SystemC has a simulation (OSCI) kernel that simulates the parallel execution of hardware. In SystemC, one can model the components of a system using *modules*. Each module can have one or more SystemC processes. SystemC uses these processes to model concurrency of hardware. The actual execution of a SystemC model is

still sequential. The OSCI kernel schedules the execution of the SystemC processes. The processes run in an interleaving manner to simulate the concurrent execution of hardware. SystemC processes can synchronize with each other using SystemC events. A process can wait for an event to be notified whereas another process can notify the event. The SystemC library, together with a single-core reference simulation kernel, is available as an open source library at http://www.accellera.org/downloads/standards/systemc.

Execution of a SystemC model consists of three phases in order: elaboration phase, simulation phase, and cleanup phase [4]. During the elaboration phase all modules are instantiated, all channels and ports are connected, and all processes are registered with the kernel. In the simulation phase, the parallel execution of SystemC processes is simulated by the kernel. The simulation is invoked by a function call to the SystemC library function *sc_start(...)* inside the *sc_main(...)* function of *main.cc*. User can provide an upper bound on the simulation time as a parameter to *sc_start(...)*. The cleanup phase can be used by the user to analyze the output of the simulation phase.

1.1.2 ABDV of SystemC models

One of the main purposes of modeling hardware-software systems using SystemC is to enable early verification of the entire system [5]. So verification of SystemC models is of great importance. There are two main approaches to verification, formal verification and dynamic verification. Let us assume that we have a model M. We want to verify if M satisfies some property P. P is also called an assertion. Formal verification checks if each possible execution of M satisfies P, whereas dynamic verification checks if a given execution of M satisfies P. Formal verification guarantees mathematical correctness of the model. But as the model size increases, formal

verification becomes practically infeasible due to state space explosion. Due to the scalability issue, most real systems are impossible to verify formally. On the other hand, dynamic verification can scale with the size of the model though it does not guarantee the correctness of the model for all possible input environments. But the coverage of dynamic verification can be increased efficiently using techniques like path coverage, automatic test generation etc. Dynamic verification can also be parallelized to test more input environments. That makes dynamic verification the most common verification technique, used to verify real systems [6].

Monitoring is one of the most common and well-known dynamic verification techniques [7]. A monitor is a program that runs with the model under verification (MUV) and verifies a particular execution of the model. Constructing monitors manually is not only a burdensome task, but also error prone. Assertion-based dynamic verification (ABDV) is a special type of monitoring where the monitors are created automatically from formal properties, called assertions [8, 9]. ABDV allows the user to verify formal assertions about the system using dynamic verification technique. There has been a lot of work in applying the ABDV approach to `SystemC` models [10, 11, 12, 13].

Here we present an overview of one such ABDV framework for `SystemC`, developed by Tabakov and Vardi [14, 10, 15]. A tool called CHIMP [16] implements this framework. In CHIMP, the user can write formal assertions about the `SystemC` model under verification using Linear Temporal Logic (LTL) [17]. An assertion states a property about the entire execution trace of the MUV. We call it a trace property. A trace property is an LTL formula interpreted over infinite trace. However, a simulation can not be run for infinite time. The simulation trace of any `SystemC` model is finite. So the monitor that is generated from a trace property can have three possible

outcomes, PASS, FAIL, and UNDETERMINED. The last case happens when there is some future obligation that is not satisfied in the finite simulation. For example, if the assertion is *"eventually p"* and p does not become true during the finite simulation, the outcome of the corresponding monitor will be UNDETERMINED. Every assertion is converted to a `C++` monitor class. During the execution of the MUV, a single instance of each monitor class is created in the elaboration phase before the simulation phase begins. The number of monitor instances is equal to the number of trace properties to verify.

1.1.3 'flow' in `SystemC`

By its nature, a SystemC model is comprised of the *components* of the system being modeled. Thus, the modeler thinks about the system *architecturally*. An orthogonal way of thinking about the system is *behaviorally* [24]. From this perspective, an execution comprises multiple units of work or "jobs", for example, load an instruction, transmit a message, handle an interrupt, and the like. Note that a single "job" can span many system components; in fact, a job often "flows" from component to component, which is why we call it a *flow*. Note that a single execution of the system can contain multiple, perhaps concurrent, flows. From this behavioral perspective, it would be natural to write and monitor assertions about flows, for example, we may want to say that every message transmission concludes successfully or gives an indication of transmission failure. Yet expressing and monitoring flow properties are quite difficult in current ABDV approaches to SystemC, whose focus is on trace properties. While it might be possible in principle to express a trace property that talks about all flows that are included in the trace, such properties would be quite unwieldy and difficult to write. To the best of our knowledge, none of the existing

framework supports monitoring of properties about flows in a SystemC model.

Let us elaborate further on the concept of flows. Most of the concurrent hardware-software systems are multi-user reactive systems. The system runs forever and different users can interact with the system during its execution. A flow can be a job submitted from outside the system or a job generated by some component of the system internally. Each flow begins with a set of input data, flows through different components in the system, and then ends producing some output data. There can be different types of flows in a system; we call them *flow types*. Example of flow types in a graphics processor are video compression, image segmentation, spatial transformation, and the like. Similarly flow types in an ATM server can be money withdrawal, check deposit, balance inquiry etc.

Each flow type is associated with a set of data variables that contain the inputs to the flow, the outputs produced by the flow, and any intermediate data. We call them *flow attributes*. For example, the flow attributes associated with 'money withdrawal' flow type are card number, pin, amount to withdraw (input attributes), transaction status (output attribute), account number, initial balance, and final balance (intermediate attributes). The flow attributes associated with *video compression* flow type of a graphics processor are *input_video*, *compression_ratio* (input attributes), *output_video*, and *output_status* (output attributes).

A flow type can be instantiated any number of times during the simulation. The instances are flows. Flows are dynamic as they can begin any time during the simulation and there can be any number of flows of each type. This is determined only in runtime. Each flow has its own values of the flow attributes, associated with its type. A flow must complete in finite time. A flow is *alive* after it begins and before it ends. A flow that is alive is also called a *live* flow. Since the system is concurrent, at

any point during the simulation there can be multiple live flows of same or different types. Each system component a flow travels through may perform some operation on some attributes of that flow.

Fig. 1.1 shows an example illustrating how flows of type *video compression* flow crosscutting different components of a graphics processor. A flow of type *video compression* begins with an input video and a desired compression ratio. The flow is initiated by some application software that wants to compress a video. Then the flow is sent to the OS, which sends it to the device driver. The device driver checks if the graphics hardware is free to do the job. If not it sets output status = BUSY and sends the flow back to the OS. The OS sends it back to the application software, at which point the flow ends. If the graphics hardware is free, the device driver sends the flow to the graphics controller, which sends it to the graphics engine. The graphics engine produces the output video and sets output status = DONE. Then it sends the flow all the way back to the graphics engine to the device driver to the OS to the application. After the application gets back the processed flow from the OS, it ends the flow. So depending on the global state of the system and the values of the input attributes, different flows of the same type can take different paths through the system components. Fig. 1.1 shows the two different paths (red and green) taken by flows of type 'video compression' depending on whether the hardware is busy or not.

Each flow type T can have one or more properties associated with it. We call them *flow properties* of type T. A flow property of type T is a temporal formula that can refer to the flow attributes of T and the global variables. A flow property describes the behavior of a *single* flow, but *all* flows of a type T must satisfy *all* flow properties of type T. An example of a flow property for the video compression flow type can be: The *output_video* should be produced within 10 clock cycles and the

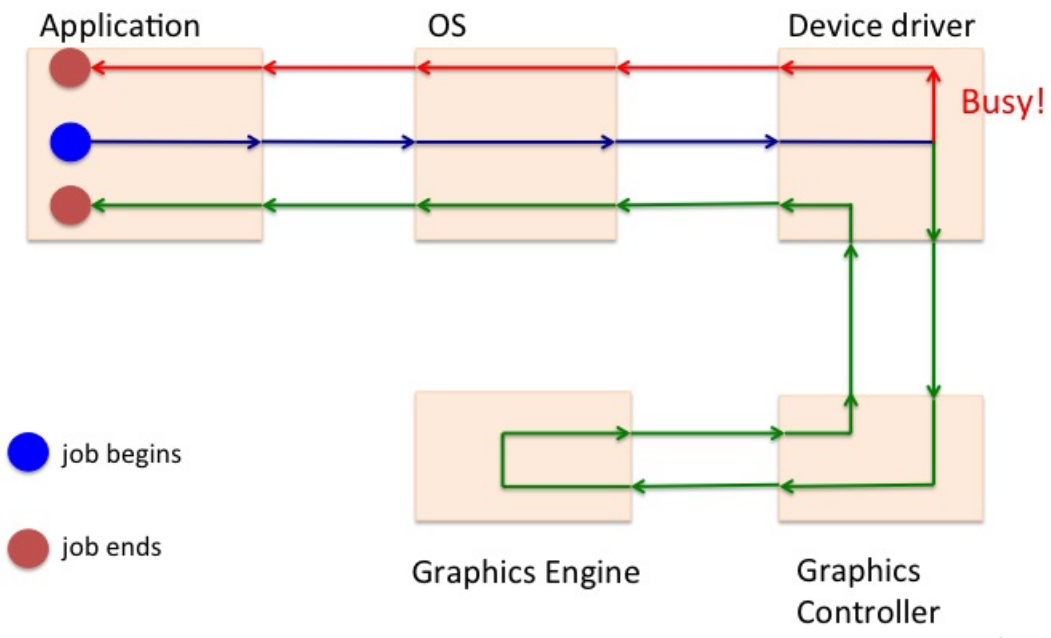

Figure 1.1 : Flows of type 'video compression' flow through different components of graphics processor

size of the *output_video* should be less than or equal to the size of the *input_video*. In this work we focus on *intra-flow* properties, where each property can refer to the attributes of only one flow at a time. Generally, intra-flow properties cannot express interaction between different flows, though they can capture the interaction between the flow and the system components by referring to global variables.

Our goal here is to enable assertion-based monitoring of flow properties. Flow properties capture the behavior of reactive systems very well. Another advantage of flow-based monitoring is that one can explore multiple behaviors of the system by generating flows with different input attribute values in a single simulation. A flow property P about a flow type T should be verified for every flow of type T. Since a flow is always finite, a flow property is interpreted over the finite trace of a flow,

unlike trace properties that are interpreted over infinite trace of the entire system. Monitoring flows in a SystemC model faces the following challenges.

First, to monitor each flow individually, the monitor has to know when a flow starts, when it ends, what its attributes are, how the attributes can be accessed, and the like. This important information has to be explicit and accessible to the monitors during the simulation. Thus, to enable flow-based monitoring, we need a methodology for explicit modeling of flows in a `SystemC` model. This includes defining different flow types and their flow attributes, beginning and ending a flow, and passing a flow from one `SystemC` process to another. In later section we show that flows can also branch and merge. We need a way to model all these features of flows in `SystemC`.

Second, the flows are dynamic and concurrent; a flow can begin anytime during the simulation and the total number of flows is not known before the simulation starts. Thus, unlike trace-property monitors, which can be instantiated before simulation starts, flow property monitors must be instantiated dynamically in the simulation phase. Thus, enabling flow-based monitoring requires the development of a software framework that that is quite different than the one that enables trace-property monitoring.

Third, a flow corresponds to a finite trace slice of the entire execution trace of the system. Each flow property should be interpreted over the finite trace slice of a flow. Since the trace slices of the simultaneously alive flows overlap, it is not clear how to extract the trace slices for each flow from the entire simulation trace. Also we need an algorithm to generate `C++` monitors from LTL formulas, interpreted over finite trace.

This thesis proposes an assertion-based flow monitoring framework for `SystemC` that addresses all the above challenges. Our framework is called Flow Monitoring

Framework. Before discussing the framework in more detail, we first look at the related work.

1.2 Related Work

The main focus of this thesis is modeling flows in `SystemC` and online monitoring of flow properties. So, the discussion of related work can be divided in two parts: work related to general modeling of flows and similar formalisms, and the work related to monitoring of flows.

1.2.1 Flow-like modeling

Schwartz-Narbonne et al. proposed Prim-Verilog [18], a language to build Transaction-level Microarchitecture Model (μ-TLM) of a hardware system. μ-TLM has an inherent notion of transaction, that enables model-checking of transaction-level properties [19, 20, 21]. The concept of transaction instances in μ-TLM is very similar to our concept of flows in `SystemC`. But our notion of flow is more general for the following four reasons; First, transactions can only begin synchronizing with a global clock whereas flows can begin anytime during the simulation; Second, at any clock cycle, no more than one instance of a transaction can begin. In contrast, any number of instances of a flow type can begin at the same clock cycle; Third, each live transaction executes one step at each clock tick, while the execution of the different steps of a live flow is completely flexible and depends on the model; And fourth, a transaction instance cannot branch and join, whereas a flow can. When two steps of a flow are independent, they can be executed by two different `SystemC` processes simultaneously. We call this "branching" of the flow. Also μ-TLM can only model hardware, not hardware-software systems. We note that a μ-TLM model can be modeled as a flow

model and properties about its transaction instances can be monitored using Flow Monitoring Framework.

Damm and Harel proposed LSC (Live Sequence Chart) [22] to describe use cases/scenarios, that crosscut the system modules [23]. Flows also crosscut the system modules. But the basic difference between LSC and flow is that LSCs can be considered as specifications about the behavior of the entire system. But flows are different jobs performed by the system and we are interested in verifying properties about flows. A new programming paradigm, Behavioral Programming [24] was introduced by Harel, Marron and Weiss, that proposed scenario-coding technique to build reactive systems incrementally from the expected behaviors, called scenarios. In behavioral programming, the user models only the scenarios and the compiler extracts the system components and their behaviors from those scenarios. But in case of modeling flows in `SystemC`, the user models both the `SystemC` modules and the flows flowing through them. This work focuses on methodology for implementing and monitoring flows in `SystemC`.

Talupur and Tuttle defined 'message flow' as a linear sequence of messages, sent among processors during the execution of a protocol [25]. Message flow is used by protocol designers to describe and reason about protocols. The work described in [25, 26] uses message flows in parameterized protocol verification by deriving high-quality protocol invariants from them. The processors can be modeled as `SystemC` processes and the message flows can be modeled as flows in a flow model. Again, the main difference between message flows and our work is that message flows only model the interaction among the different components of the system. But we model both the system components and the flows flowing through the components in a `SystemC` model. In our case there is no separate model for flows unlike LSCs, Behavioral

Programming or 'message flows'.

A workflow [27] consists of a sequence of connected steps, where each step follows without delay or gap and ends before the subsequent step may begin. Workflow management system aims at modeling and controlling the execution of processes in business, scientific, or even engineering applications [28]. Workflow languages [28, 29] aim at capturing workflow-relevant information of application processes with the aim of their controlled execution by a workflow management system. Our `SystemC` flows can be described as domain-specific workflows, the domain being `SystemC`.

All of the above work focus on modeling flows; in contrast, architectural models focus on modeling components of the system. Stitching together flow models and architectural models is a major challenge. Aspect-oriented programming [35] is one way of stitching together architectural models with crosscutting concerns, but aspect-oriented programming by itself is not rich enough to model flows. For example, Maoz, Harel, and Kleinbort describe a compiler for transforming LSCs into AspectJ [23]. In contrast, our focus is on enabling users to build SystemC models that are simultaneously both architectural models and flow models. The underlying philosophy of our approach is that flows are already implicitly present in the architectural models, since the components of the system do have to perform the operations of the flows; all that is needed is a thin layer of annotation to make these flows explicit, which enables monitoring their properties. Thus, rather than building separate architectural models and flow models and then attempting to stitch them together, users can directly build architectural flow models.

1.2.2 Monitoring of flows

We now describe the two most relevant pieces of prior work, related to monitoring of flows: MOP (Monitor-Oriented Programming) and SCV (SystemC Verification Library). MOP [7] does parametric trace slicing and monitoring [30] for of assertions about programs written in Java. In Java, one can create an enumerator of a vector to iterate over the elements of that vector. Example of a parametric property is: *a vector should not change when one of its enumerators is used*. This property is verified for all the vectors in the Java program under test. MOP does dynamic and decentralized monitoring. 'Decentralized' means that one monitor instance is generated for each vector and enumeration pair. 'Dynamic' implies that the monitor instances are generated and destroyed dynamically during the execution. Our Flow Monitoring Framework is dynamic and decentralized, influenced by MOP. The main difference between our work and MOP is that MOP does not have a notion of cross-cutting flows across processes. Further, flows are related to hardware-software concurrent systems, whereas Java is a software modeling language. Being a system modeling language, `SystemC` has its own unique simulation semantics, very different from Java.

SCV [31] is developed by the Verification Working Group (VWG) of `SystemC` to provide a complete platform for verification of `SystemC` models. SCV provides a set of APIs that are used as a basis for verification activities with `SystemC` such as, transaction recording, generation of random values under constraints etc. SCV allows users to define transaction types and capture transaction level activities during simulation. These activities can be monitored by another SystemC module at runtime through a callback mechanism or can be recorded into a database for visualization, debugging, and post simulation analysis. But SCV does not have a notion of crosscutting flows that can branch and join. Also our Flow Monitoring Framework is very light-weight

compared to SCV in terms of size of the framework and coding overhead.

1.3 Contribution

The primary contribution of this thesis is a methodology for efficient implementation of flows in a `SystemC` model and monitor flow properties about those flows. This includes the following three pieces.

1.3.1 Flow Library

We define the concept of flows in SystemC, which leads to flow-based SystemC models (*flow models*, for short). We believe that the idea of flows is already implicit in many SystemC models, but we want to "expose" flows and make them explicit. Furthermore, our goal is to accomplish that with a minimum amount of annotation. For that, we provide a light-weight (228 LOC), yet robust and efficient `C++` library, called *Flow Library*, using which one can either design a flow model from the scratch or annotate flows in an existing SystemC model with minimal change. Using Flow Library one can model different flow types in a `SystemC` model, each having its own set of flow attributes. The Flow Library allows the user to begin a new flow, end an alive flow, and transfer a flow from one `SystemC` process to another. The Flow Library is designed in such a way that very little (one line) modification of the flow model is needed to execute it with and without monitors, with the guarantee that the non-monitored execution does not incur any extra overhead of monitoring. Chapter 2 describes this library in detail.

1.3.2 Flow Algorithms

This thesis presents two algorithms for monitoring flow properties: Flow Monitor Generation Algorithm and Flow Monitoring Algorithm. FlowMonGen algorithm describes how to generate a `C++` flow monitor class from a flow property. Since flows are finite, a flow property is an LTL formula interpreted over the finite trace of a flow, which is a finite trace slice of the entire execution trace. Our tool FlowMonGen implements our Flow Monitor Generation Algorithm and can be used to generate flow monitor classes from flow properties automatically. The complexity of Flow Monitor Generation Algorithm is doubly exponential in the size of the input flow property. But in practice we rarely experience the worst case complexity. The Flow Monitor Theorem (stated in Chapter 3) guarantees that the flow monitors generated by FlowMonGen are correct and the monitor output is 2-valued: PASS and FAIL for all complete flows.

The second algorithm is a dynamic and decentralized algorithm that describes when to create and delete flow monitor instances and when to execute them. We call it Flow Monitoring Algorithm. This algorithm is dynamic because flow monitor instances are created dynamically in the simulation phase synchronizing with the beginning of new flows. It is decentralized because there is one flow monitor instance per flow and flow property pair (as opposed to, one monitor instance per trace property). This algorithm is implemented in our Flow Monitoring Framework, discussed in Chapter 4.

1.3.3 Flow Monitoring Framework

This is a complete and fully automated framework to verify flow properties associated with different flow types of a flow model. This framework consists of three compo-

nents, the Flow Library, a flow model (MUV), and the flow monitors automatically generated from the flow properties using the FlowMonGen tool. It implements Flow Monitoring Algorithm including how to automatically instantiate and delete a flow monitor instance, how and when to execute the flow monitors. It allows the user to execute the flow monitors at different levels of abstraction: low levels like every time a flow attribute changes its value to high-levels like kernel phases and `SystemC` process suspensions.

The Flow Library and the FlowMonGen tool are available as a open source software package, called SystemC Flow Package. The SystemC Flow Package can be downloaded at `https://sourceforge.net/projects/SystemCFlow`.

1.4 Organization

The remainder of this thesis is organized as follows. Chapter 2 defines the different terminologies related to flows in a `SystemC` model with concrete examples. It also describes the Flow Library and how to use it to design a flow model. Chapter 3 presents the algorithms related to flow monitoring, the Flow Monitor Generation Algorithm and the Flow Monitoring Algorithm. In Chapter 4, we present the complete Flow Monitoring Framework and how different pieces work together to monitor the flow properties in a flow model. Chapter 5 describes a case study, used to analyze the performance of this framework and the experimental results. Finally, Chapter 6 concludes this thesis and discusses the future work.

Chapter 2

Flow Library

2.1 flows in SystemC

In this section we define the different terminologies related to flows in `SystemC` with some concrete examples.

2.1.1 flow types

There can be different types of flows in a flow model. Each type of flow is entitled to do a specific job. We call them flow types.

Example 2.1 In an ATM server, there can be the following three flow types: 'money withdrawal', 'check deposit', and 'balance inquiry'. These three flow types represent three different types of jobs a user can perform at an ATM machine, such as withdrawing certain amount of money from a bank account, depositing a check to an account, inquiring the current balance of an account respectively.

2.1.2 flow attributes

Each flow type is associated with a set of data variables, called flow attributes. These variables contain the inputs to the flow (input attributes), the outputs generated by the flow (output attributes) and any intermediate data (intermediate attributes). The same attribute can also act as both input and output attribute by taking the input value when a flow begins and then producing the output value before the flow ends.

Example 2.2 The flow attributes associated with 'money withdrawal flow type are card number, pin, amount to withdraw (input attributes), transaction status (output attribute), account number, initial balance, and final balance (intermediate attributes).

2.1.3 flows

A flow is an instance of a flow type. A flow type can be instantiated any number of times during the simulation. The number depends on how the user(s) are modeled in the flow model and how long the simulation is run. Since the system is concurrent, multiple flows of same or different types can begin and flow through the system at the same time. Each flow of a flow type has its own values of the flow attributes related to that flow type. A flow does the following during the simulation:

1. Begins with a set of input data (assigned to the input attributes of the flow).
2. Flows from one `SystemC` process to another. These `SystemC` processes read and write the flow attributes to produce the values of the output attributes. They also update the global state of the system accordingly.
3. The flow ends after finite time.

A flow is **alive** after it begins and before it ends. A flow that is alive is also called *live* flow. In a concurrent system, multiple flows of same or different types can be alive at the same time during the simulation.

Example 2.3 We present here a detailed example showing how the flows of a flow type flow through the components of a system. Fig 2.1* shows the different possible

*The figure is best viewed online.

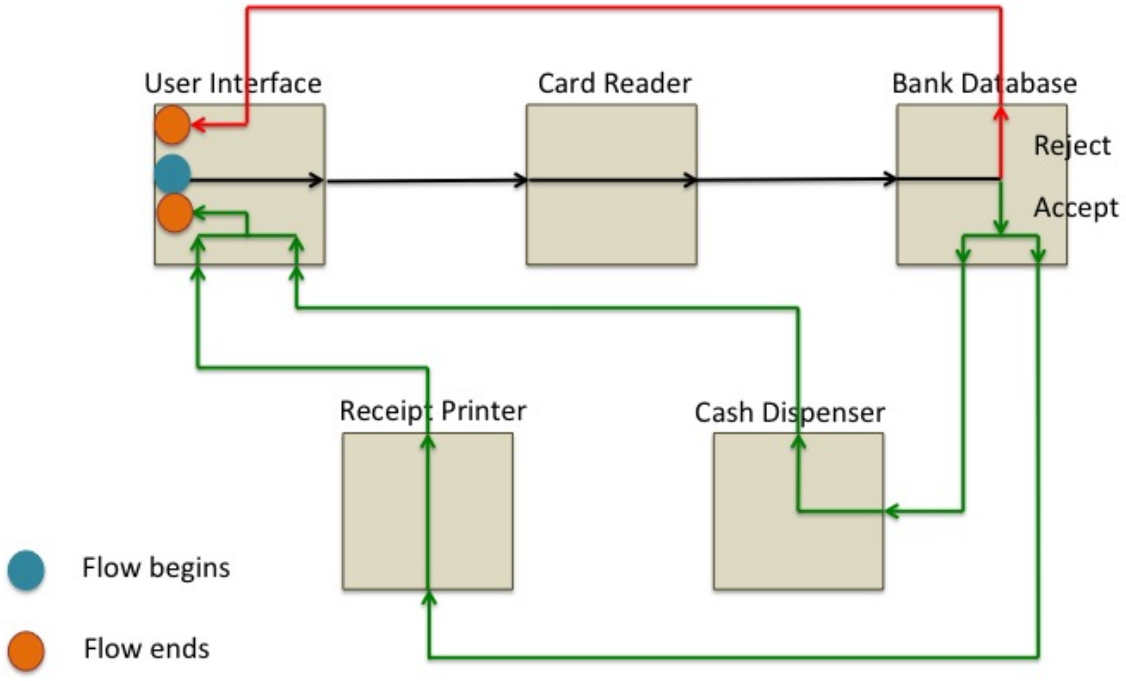

Figure 2.1 : Paths taken by flows of type ‘money withdrawal’ in ATM server

paths that a flow of type 'money withdrawal' in an ATM server may take. A 'money withdrawal' flow begins in a user interface, when a user chooses to withdraw money through an ATM machine connected to that ATM server. The flow then goes to the card reader, where the three input attribute values (*card number*, *pin* and *amount to withdraw*) are assigned. Then the flow moves to the bank database, which checks if the card is valid, the *pin* is correct, and the *amount to withdraw* is less than the current balance in bank database. If any of these checks fails, the output *transaction status* is assigned REJECT and the flow goes back to the user interface and ends. Else, the current balance is updated in the database and the *transaction status* is assigned to ACCEPT. Now the flow concurrent goes to both cash dispenser and receipt printer to give out the cash and print the receipt respectively. Note that the operations done by the cash dispenser and the receipt printer on the flow are independent of each other and can be done in parallel. So in a SystemC simulation, these two operations can be interleaved in any order. This is called *branching* of a flow. Once a flow branches, it must merge before it ends to avoid unexpected behaviors. An example of such unexpected behavior in our case is when the cash dispenser ends the flow while the receipt printer is still processing it. So a branching should be followed by a merge eventually. In this case, the flow merges at user interface and ends.

Different flows of a flow type can take different paths through the system depending on the current state of the system and input attributes values. For example, in Fig. 2.1, if the input pin is incorrect, the flow takes the red path, otherwise it takes the green path. Irrespective of what path a flow takes, it must satisfy all the flow properties associated with the money-withdrawal flow type. One such flow property could be: "if the card is valid, the *pin* is correct, and the *amount to withdraw* is less than the current balance, then eventually globally *transaction status* will be AC-

CEPT; else eventually globally it will be REJECT [†]. Note that the current balance here is a global variable, stored in the bank database.

2.1.4 flow properties

A flow property is an LTL (Linear Temporal Logic) [17] formula that describes the flows of a flow type. There are two types of flow properties: intra-flow properties and inter-flow properties. An intra-flow property can only refer to the attributes of a single flow, whereas an inter-flow property refers to more than one flow. This work only focuses on intra-flow properties. From now on, we mean only intra-flow properties by the term 'flow properties'. Each flow type T has a set of flow properties associated with it. We call them flow properties of type T. A flow property of type T can refer to the flow attributes of T and the global variables of the SystemC model. A flow property of type T must be satisfied by all flows of type T.

Example 2.4 Let us assume that the bank database of the ATM server contains a map **M** containing the information about all the accounts. The key of **M** is the account number and the value is an object containing different fields like account owner's name, balance etc. A flow property of type 'money withdrawal' of ATM server can be: If the *pin* is correct and the *initial balance* is greater than or equal to the *amount to withdraw*, then eventually *final balance* and the balance field of M[*account number*] becomes equal to (*initial balance* - *amount to withdraw*) and eventually after that *transaction status* becomes "ACCEPT"; Else eventually *transaction status* becomes "REJECT" and *final balance* becomes equal to initial balance.

[†]We use Linear Temporal Logic to specify flow properties; see below.

2.2 Flow Library

The first contribution of this thesis is a light-weight `C++` library (228 LOC), called Flow Library, using which one can design flow models from scratch or annotate flows in an existing SystemC model. This section describes different components of Flow Library and how to use this library to design a flow model. Flow Library provides base classes and APIs for modeling different types of flows, beginning and ending of a flow, and transferring of a flow from one `SystemC` process to another. All these aspects are modeled keeping the ease of monitoring of flows in mind. The monitors should be automatically informed whenever a new flow begins or a live flow ends. The monitors should also be able to access the flow attributes of any flow efficiently during the monitored simulation.

2.2.1 Components

Flow Library contains three classes written in five files: `flow class` in *flow.h* and *flow.cc* files, `flow_manager class` in *flow_manager.h* and *flow_manager.cc* files, and `base_monitor class` in *base_monitor.h* file. It also defines an enum, called *error_code*, in flow_manager.h that contains a list of error codes that the APIs of Flow Library return. The error codes enable some automatic error checking on the implementation of flows in flow models.

enum error_code

Flow Library defines a set of possible error codes to enforce some basic properties of flows on the flows of a flow model. Each API defined in the Flow Library returns an error_code indicating if the input argument is valid or the user is trying to do some invalid operation. For example, a flow can begin only once; a flow cannot be referred

Error name	Cause of occurance
DONE	No error occurred.
NOT_ALIVE	A process is trying to access a flow that is not alive.
CANNOT_END	A process is trying to end a flow that is being used by some other process.
ALREADY_ALIVE	A process is trying to begin a flow that is already alive.
CANNOT_RELEASE	A process is trying to release the access of a flow too many times.
NULL_FLOW	The flow pointer is NULL.

Table 2.1 : Error codes defined in enum *error_code*

or accessed if it is is not a live flow; a process cannot end a flow if some other process is still using it etc. All the error codes are defined as an enum, called *error_code* in *flow_manaher.h* file. We will see the use of these error code later in this section. Table 2.1 shows all the error codes defined in Flow Library. All these codes other than *DONE* indicate erroneous implementation of flows in the flow model and must be corrected.

flow class

Given that each flow type has a set of flow attributes and a flow is just an instance of a flow type, it is easy to see that the most object-oriented way to model flows is to model flow types as classes and flow attributes as their member variables. Then a flow is nothing but an object of its flow type class. Also beginning and ending a flow become as simple as creating an object of its flow type class and deleting that object.

But there is some common data associated with all flows irrespective of their types. For example, to identify each flow uniquely, each flow object needs to have a

unique id. So Flow Library provides a base class for all the user-defined flow type classes, called *flow class*. The `flow class` has one static member variable: static unsigned int *new_flow_id*, and three non-static member variables: const unsigned int *flow_id*, const int *flow_type* and unsigned int *num_proc*.

The member variable *flow_id* contains the unique id of each flow. It is the job of the Flow Library to assign a unique value to the *flow_id* of every newly begun flow. The static member variable *new_flow_id* is used by the `flow class` to assign unique id to every flow object. Everytimne a new flow begins, the value of *new_flow_id* is incremented by 1 and is assigned to the *flow_id* of that newly begun flow.

There is a unique id associated with each user-defined flow type classes, called flow type id. If there are n user-defined flow type classes in a flow model, the flow type ids will vary from 0 to $n-1$. These ids are decided by the designer of the flow model and passed to the `flow class` while constructing flow objects. Each flow object contains the unique id of its flow type in a member variable called *flow_type*. All flows of the same flow type have the same value of *flow_type*.

Since a flow can branch, a flow can be processed by more than one `SystemC` process at the same time. To avoid accidental ending of a flow, it is important to keep track of the number of processes processing a flow at any point in time. Then if a process tries to end a flow when some other process is still operating on that flow, the Flow Library can raise an error. At any time during the simulation, *num_proc* member variable of a flow contains the number of processes operating on the flow. *num_proc* is initialized to 1 inside the constructor of `flow class` when the flow begins. The value of *num_proc* is updated automatically by the Flow Library. The *num_proc* of flow f is incremented by 1 every time a process starts working on f and is decremented by 1 every time a process finishes working on f. *flow_id* and *flow_type* are declared

as *const* and initialized in the flow class constructor when a flow begins.

`flow class` contains three getter functions for *flow_id*, *flow_type*, and *num_proc*. The `flow class` also contains two more functions: *void flow::increment_num_proc()* that increments *num_proc* by one whenever a `SystemC` process starts working on the flow; and *error_code flow::decrement_num_proc()* that decrements *num_proc* by one whenever a `SystemC` process finishes working on the flow. The second function returns *error_code::CANNOT_RELEASE* if the *num_proc* is already 0. Otherwise it decrements *num_proc* and returns *error_code::DONE*.

flow_manager class

To monitor flows, the monitors need to know when a new flow begins, when it ends, how to access its attributes and the like. The cleanest way to expose this information to the monitors is to have a single point of access (independent of the user-defined flow types) in the Flow Library, through which the monitors can access information about all flows without interacting with every flow directly. For that Flow Library provides a `flow_manager class` that keeps all the information about the flows and provides it to the flow monitors as necessary. `flow_manager class` maintains a data structure, called *alive_flows* that contains the list of all live flows.

The `flow_manager class` provides four APIs that can be used by the `SystemC` processes of the flow model to register flows with flow_manager and access those flows from anywhere in the model. They are:

begin_flow : used by a process to begin a new flow.

end_flow : used by a process to end a live flow.

get_flow : used to access a flow object, given its flow_id.

release_flow : For security purposes, the flow_manager keeps track of how many processes are working on a flow at the same time (in the member variable *num_proc*). So, when a process is done working on a flow, it calls release_flow to notify the flow_manager about it.

Each of the these four APIs returns an error code.

In `SystemC` there can be two types of processes: thread process and method process. The thread processes execute only once during the simulation. But they can suspend and resume themselves during the simulation, as many times as they want. Normally thread processes are designed to have a infinite while loop that is synchronized with some clock. At the end of each iteration, the thread process suspends itself and waits for the next clock cycle. The method processes works a bit differently. A method process can execute multiple times during the simulation. Unlike thread processes, method processes cannot suspend themselves. Usually method processes have a static sensitivity to some clock. At every cycle of that clock, the method process executes once. So both method and thread process meet the same purpose of repeating sequence of actions. The only difference is the way they are written.

If a process P (method or thread) operates on clock C, it performs a sequence of actions at every clock cycle of C. Assume that P is a process in a flow model. For example, P is a process in the bank database of the ATM server that runs the validity check on each transaction. So, at every clock cycle of C, P operates on some flow, the flows that it obtains from the card reader. So a process performs the following three steps in sequence at every cycle of the clock it operates on:

1. Either begins a new flow (if it is a user process or a application process that submits jobs) or gets a flow from some other process.

2. Reads and writes the attributes of that flow and update the state of the system accordingly.

3. Either sends the flow in some other process or ends the flow.

The `flow_manager class` also does some error checking and returns appropriate error codes (defined in *error_code*) to the user method that makes a call to `flow_manager class`. Another member variable in `flow_manager class` is const int *num_flow_types* that denotes the number of user-defined flow types in the flow model. This variable is assigned in the constructor of the `flow_manager class`. Each flow model defines a global pointer variable *flow_manager* fmanager* that points to an object of flow_manager class. *fmanager* is used to register and access flows throughout the flow model.

base_monitor class

The last class is `base_monitor class`. This class is not needed to write a flow model. This class is used during monitoring of flows. `base_monitor class` serves as the base class for all the monitor classes. The user does not have to instantiate or use this class anywhere in his flow model. We will see this class's functionality in Chapter 4 in detail.

2.2.2 Modeling flow types and flow attributes

A user-defined flow type T in a flow model can be defined as a `C++` class T, derived from the `flow class`. The flow attributes of flow type T can be modeled as member variables of class T. Each user-defined flow type should have a unique integer id. This id should vary from 0 to (number of user-defined flow types $-$ 1). It is the

responsibility of the user to assign an unique id to each user-defined flow type. In the constructor of the flow type class, T for example, the base class constructor should be called with id of flow type T as parameter.

A flow type class constructor must initialize all the flow attributes. If it is an input attribute, the value will be passed by the process that is creating this flow. The values of the output and intermediate attributes should be set to some default values. The flow attributes are recommended to be modeled as private member variables of the flow type class and to be accessed only through setter and getter methods.

The flow model of ATM server in Fig. 2.1 has three flow types: 'money withdrawal', 'check deposit', and 'balance inquiry'. Let us assume that the unique ids of these flow types are 0, 1, and 2 respectively. Listing below shows how to model the class for 'money withdrawal' flow type.

```
/**
* Class representing flow type 'withdraw money'
*/
#ifndef WITHDRAW_MONEY_H
#define WITHDRAW_MONEY_H
#include "flow.h"
//Other files to include if needed
...
enum trans_status{NOT_ASSIGNED, ACCEPT, REJECT};
class withdraw_money : public flow{
      public:     //Constructor
        withdraw_money(unsigned int amount) :
        /*type_id of this flow_type = 0*/
```

```
        flow(0),
        card_number(""),
        pin(""),
        amount_to_withdraw(amount),
        status(NOT_ASSIGNED),
        cash_given(false),
        account_number(""),
        initial_balance(0),
        final_balance(0)
        {}//End of constructor
        //Setters and getters for the flow attributes
        ...
        //Other functions if needed
        ...
    private: //Flow attributes
        std::string card_number;
        std::string pin;
        unsigned int amount_to_withdraw;
        trans_status status;
        bool cash_given;
        std::string account_number;
        unsigned int initial_balance;
        unsigned int final_balance;
}; //End of class
```

2.2.3 Beginning and ending a flow

In a flow model, one or more `SystemC` processes can be responsible to generate new flows. Such `SystemC` process represents either a user of the system or some application that is submitting new jobs to the system. As discussed earlier, this can be a thread or a method process operating on some clock C, defined in the flow model. At every clock cycle, the process may begin a new flow by instantiating a flow type class. To begin a flow of type T, a process does the following:

```
T* f=new T(<values of input attributes>);
error_code e = fmanager->begin_flow(f);
```

begin_flow function is defined in the `flow_manager class` as *flow* begin_flow(flow* f, error_code* e)*. In this function f is the pointer to the newly begun flow object and e is the error_code generated by this function. The return value is the same as f to enhance the usability of this API. *begin_flow* checks if f is indeed a new flow. If not it returns ALREADY_ALIVE. If f is a null pointer, it returns NULL_FLOW. Else it inserts f in *alive_flows* database and returns DONE. The following Listing shows how to model the process in User Interface module of ATM that begins 'money_withdrawal' flows and send them to a process of card reader module.

```
void user_interface::submit_requests(){
  while(1){
    /*Randomly decide which type of request
    to submit. Options are withdraw money,
    deposit check and inquire balance.*/
    ...
    if(request_type == "withdraw_money"){
```

```
      /*choose the amount to withdraw*/
       unsigned int amt = ... ;

       //Begin a "withdraw money" flow
       error_code e;
       withdraw_moey* f=new withdraw_money(amt);
       error_code e = fmanager->begin_flow(f);
       assert(e == DONE); //Or other actions

       /*Put f->get_flow_id() in a FIFO
       channel to card_reader*/
       ...
       /*Done operating on flow f.
       Release f. (Details later)*/
       error_code e1=fmanager->release_flow();
       ..
       //Wait for the next clock cycle
       //Statically sensitive to user_clock
       wait();
     } //End of if
     elseif
       ...
  } //End of while(1)
} //End of thread process
```

Similarly a process can end a flow after operating on it. To end a flow f, a process calls *end_flow* with the id of f: fid.

```
error_code e = fmanager->end_flow(fid);
```

end_flow function is defined in the `flow_manager class` as *error_code end_flow(unsigned int fid)*. In this function fid is the unique *flow_id* of the flow to end, f. If fid is not the flow_id of an alive flow, end_flow returns NOT_ALIVE. If some other process is still using the flow, it returns CANNOT_END. Otherwise, end_flow removes f from alive_flows, deallocates the memory of f, and returns DONE. The following Listing shows how to model the process in User Interface module of ATM server that ends flows of type 'money_withdrawal' that did not get rejected at the bank database. Notice that ending also involves merging of the flow from receipt printer and cash dispenser.

```
void user_interface::end_withdraw_money_success(){
  while(1){
    /*Get the id of the next flow
    sent by receipt_printer*/
    unsigned int id = ... ;
    /*Get the corresponding flow pointer*/
    withdraw_money* f = fmanager->get_flow(id) ;

    /*Merge the flow from cash
    dispenser and receipt printer*/
    if(!f->get_cash_given()){
        /*Wait until cash_dispenser is
```

```
        done processing this flow. Can be
        modeled using SystemC event.*/
        wait(...);
    } /*If the flow is still being used by
    cash_dispenser*/

    //Process the flow
    f->set_status(SUCCESS);

    //Now end the flow
    error_code e = fmanager->end_flow(id);
    assert(e == DONE);

    //Wait for next clock cycle
    wait();//Static sensitivity
  } //End of while(1)
} //End of thread process
```

2.2.4 Transferring a flow from one `SystemC` process to another

A flow travels from one process to another through shared elements. The shared elements can be any global FIFO queues. The size of these queues will put a bound on the maximum number of live flows in the system. If the two processes involved in transferring a flow are in the same `SystemC` module, the shared element can be a member variable of the module. If the two processes are in two different `SystemC` modules, the flow can be transferred through `SystemC` FIFO channels. It is the choice

of the designer how to transfer a flow from one `SystemC` process to another.

To avoid unintended mishandling of pointers, each process should only transfer the flow_id of a flow through shared elements, not the entire flow pointer. When a process receives a flow_id from some shared element, it gains access to the corresponding flow pointer using the *get_flow* API of `flow_manager class` as:

```
//f is the flow whose access is needed.
//fid: id of f; T: type of f
error_code e;
T* f = (T*)fmanager->get_flow(fid, &e);
```

get_flow is defined in the `flow_manager class` as *flow* flow_manager::get_flow(unsigned int fid, error_code* e)*, where the fid is the *flow_id* of the flow (f) received and e is the error_code that is assigned its value in this function. *get_flow* assigns NOT_ALIVE to e and returns 0 if f is not alive. Else it increment f->num_proc by one, assigns DONE to e and returns pointer to f.

At every clock cycle, a process either begins a new flow or fetch the id of a flow from some shared element. Then it access the actual flow object from *fmanager* using that id. Once it obtains the access of the actual flow object, it starts operating on the attributes of that flow and updates the global state of the system accordingly. After the process finishes operating on that flow, it either ends the flow or send the flow to one (transfer) or more (branching) processes.

Before sending a flow to another process, a process must release the flow by calling the function *error_code flow_manager::release_flow(unsigned int fid)* as below. Here fid is the *flow_id* of the corresponding flow f.

```
error_code e=fmanager->release_flow(fid);
```

release_flow returns NOT_ALIVE if f has already ended. Else it decrements f->num_proc by one and returns DONE. If num_proc is already 0, it returns CANNOT_RELEASE instead. The following Listing shows how the process card_reader::read_card() of ATM server gets a flow from user_interface::submit_request() process and sends it to bank_database::get_request() process.

```
void card_reader::read_card(){
  while(1){
    /*Fetch the id of the next flow
    sent by user_interface module.*/
    unsigned int id = ... ;
    //Get the corresponding flow pointer
    error_code e;
    withdraw_money* f = (withdraw_money*)
        manager->get_flow(id,&e);
    assert(e == DONE);

    //Process the flow
    /*randomly pick a card number*/
    f->set_card_number(...);
    /*assign correct pin with
    probability 0.8*/
    f->set_pin(...);

    /*Transfer id to bank_database
    ::receive_flow() process*/
```

```
    ...

    //Release the flow
    e = fmanager->release_flow(id);
    assert(e == DONE);

    //Wait for next clock cycle
    wait(); //Static sensitivity
  } //End of while(1)
} //End of thread process
```

2.2.5 Branching and joining a flow

A flow may branch when two operations on the flow are independent and can be done by two different processes concurrently without affecting each other's operation. This means when more than one process operate on a single flow object concurrently, they writes to different sets of variables. Since the steps are independent, they ideally should not write to the same flow attribute and there will be no race condition. If they do, it is the responsibility of the user to avoid race condition by implementing locking or some similar mechanism to avoid it.

The Flow Library does not provide any direct API for branching and joining of the flows. But they can be done by following the methodology described below. Branching of a flow may happen when two or more steps of the flow are independent of each other and can be done simultaneously by two or more processes (Let's call them branch processes). The process that branches a flow, sends the flow_id to all the branch processes in any order. Now all the branch processes start operating on

the flow simultaneously. During branching if any of the processes tries to end the flow while other processes are still working on the flow, the flow_manager returns *error_dode::CANNOT_END*.

In Fig. 2.1, the 'money_withdrawal' flow branches in the bank database and goes to both receipt printer and cash dispenser. The following Listing shows how to model the process bank_database::receive_flow() that receives a flow from card_reader and then either sends it to IO module upon transaction failure or branches it and sends it to both receipt printer and cash dispenser upon success.

```
void bank_database::receive_flow(){
  while(1){
    /*Fetch the id and
    get the flow pointer*/
    unsigned int id = ... ;
    withdraw_money* f = ... ;

    if(!((f->get_account_number()).valid())
     || ...) {
      //send f back to user_interface
    } //End of if

    /*Else branch f to receipt
    printer and cash dispenser.*/
    else{
      /*Put id in the channel
      to receipt printer*/
```

```
        ...
        /*Put id in the channel
        to cash dispenser*/
        ...
      } //End of else

      error_code e = manager->release_flow(id);
      assert(e == DONE);
      ...

    //Wait for next clock cycle
      wait(); //Static sensitivity
   } //End of while(1)
} //End of thread process
```

Join can happen only after all the branch processes finish operating on the flow. The process where the flow joins, must wait until then. This synchronization can be easily implemented using SystemC events and flow attributes. Refer to the end_flow Listing above to see how a 'money withdrawal' flow joins in the user module. For example, in Listing 2.2.3, the 'money withdrawal' flow joins in user_interface::end_withdraw_money() process. In this case the branch processes are receipt_printer::print_receipt() and cash_dispenser::give_cash(). user_interface::end_withdraw_money() process receives the *flow_id* from receipt printer and then checks if the cash dispenser is done operating on the flow. If not, it waits until cash dispenser notifies an event saying so. Once the event is notified, the end_withdraw_money process ends the flow. Thus join is implicit and must happen after the branch and before the flow ends.

Chapter 3

Flow Algorithms

In this section we present two main algorithms related to monitoring of flows in a flow model. One is Flow Monitor Generation Algorithm that describes how to generate a C++flow monitor class from a flow property. The other one is Flow Monitoring Algorithm that describes when to instantiate these flow monitor classes and when and how to execute those flow monitor instances. We shall also present the complexity, proof of correctness, and the implementation references of these two algorithms in this section.

3.1 Finite trace slice of a flow

By definition a flow is finite. The trace of a flow is a finite trace slice of the entire execution trace of a flow model. The trace slice of flow f begins when the flow f begins during the simulation and ends when the flow f ends. Since there can be multiple flows alive at the same time during the simulation, the trace slices of simultaneously live flows overlap. Fig. 3.1 shows the overlapping finite traces π_{f1}, π_{f2}, and π_{f3} of three concurrent flows f_1, f_2, and f_3 as trace slices of the entire execution trace π.

A flow property is interpreted over the finite trace slice of a flow. So a flow property is an LTL formula interpreted over finite trace. They are defined as LTL_f formulas by Giacomo and Vardi in [32]. LTL and LTL_f formulas have the exact same syntax but different semantics as described later.

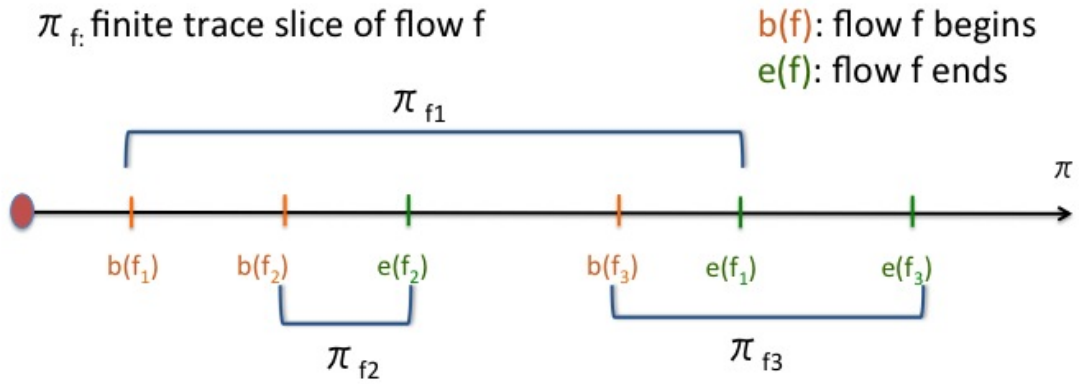

Figure 3.1 : Showing three overlapping trace slices π_{f1}, π_{f2}, and π_{f3} for flows f_1, f_2, and f_3 respectively, π being the entire simulation trace

Definition 3.1 A flow f of type T satisfies flow property P, associated with flow type T iff the finite trace slice π_f of f finitely satisfies the LTL_f formula P.

Definition 3.2 The satisfiability of an LTL_f formula φ_f by a finite trace $\pi(0 \cdots n)$ is inductively defined as:

- $\pi(0 \cdots n) \models_f p$ iff $p \in \pi(0)$, where p is an atomic proposition.
- $\pi(0 \cdots n) \models_f (\neg\psi)$ iff $\pi(0 \cdots n) \not\models_f \psi$.
- $\pi(0 \cdots n) \models_f (\varphi_1 \wedge \varphi_2)$ iff $\pi(0 \cdots n) \models_f \varphi_1$ and $\pi(0 \cdots n) \models_f \varphi_2$.
- $\pi(0 \cdots n) \models_f (X\psi)$ iff $n \geq 1$ and $\pi(1 \cdots n) \models_f \psi$.
- $\pi(0 \cdots n) \models_f (\varphi_1 U \varphi_2)$ iff $\exists j \leq n$ such that $\pi(j \cdots n) \models_f \varphi_2$ and $\forall k\ 0 \leq k < j$, $\pi(k \cdots n) \models_f \varphi_1$

It is clear from the above definition that in the semantics of LTL_f, all the future obligations have to be met before the trace ends. So given a finite trace $\pi(0 \cdots n)$ and

an LTL_f formula φ_f, whether $\pi(0 \cdots n) \models_f \varphi_f$ can be determined with certainty in finite time, more specifically no later than immediately after $\pi(0 \cdots n)$ ends. So unlike monitoring of trace property, monitoring of flow property does not have the outcome UNDETERMINED for any completed flow. We do not monitor incomplete flows, flows that did not end during the simulation. Rather we report all the incomplete flows at the end of the simulation.

3.2 Flow Monitor Generation Algorithm

3.2.1 Algorithm

Definition 3.3 A finite word x over alphabet Σ is a **bad prefix** of language L iff for all infinite word $y \in \Sigma^\omega$, $x.y \notin L$.

In [14], Tabakov, Rozier and Vardi propose an algorithm to generate a `C++` monitor class M from a trace property φ (LTL formula) such that an instance of M rejects precisely all the minimal bad prefixes of φ. M is nothing but a `C++` encoding of the DFW (Deterministic Finite Automaton on Word) that rejects all the minimal bad prefixes of φ. [33]

Theorem 3.1 An instance of monitor M rejects precisely the minimal bad prefixes of φ. [14]

A very efficient implementation of LTL to DFW(that rejects all minimal bad prefixes of the LTL formula) transformation is available in an open source `C++` library, called SPOT [34]. Our goal is to build a `C++` monitor class M_f from the flow property φ_f (LTL_f formula) such that an instance of M_f accepts flow f iff the trace slice of f finitely satisfies φ_f. Keeping the implementation efficiency in mind, we use

SPOT's LTL to DFW transformation to do our LTL_f to DFW transformation. For that we first preprocess φ_f using the transformation function $t : LTL_f \rightarrow LTL$ as defined below to generate an LTL formula φ. This transformation is first proposed by Giacomo and Vardi as reduction of LTL_f satisfiability to LTL satisfiability in [32]. Then we apply the LTL to C++ monitor generation algorithm as given in [14] on the post-processed formula φ to generate a monitor class M. Later we prove that an instance of M accepts flow f iff the trace slice of flow f finitely satisfies φ_f.

Definition 3.4 To define $g(\varphi_f)$, we introduce a new atomic proposition **alive** that does not occur in φ_f. We define the transformation function $g : LTL_f \rightarrow LTL$ as $g(\varphi_f) = t(\varphi_f) \wedge (\textbf{alive}\ U\ G!\textbf{alive})$. The function $t : LTL_f \rightarrow LTL$ is inductively defined as follows:

- $t(p) = p$ where p is an atomic proposition.
- $t(\neg\psi) = \neg t(\psi)$
- $t(\varphi_1 \wedge \varphi_2) = t(\varphi_1) \wedge t(\varphi_2)$
- $t(X\varphi) = X(\textbf{alive} \wedge t(\varphi))$
- $t(\varphi_1 U \varphi_2) = t(\varphi_1) U (\textbf{alive} \wedge t(\varphi_2))$

Intuitively atomic proposition **alive** is true until the finite trace does not end and once the trace ends, **alive** becomes false and never changes again. This expected behavior of **alive** is modeled as ($\textbf{alive}\ U\ G!\textbf{alive}$), which is used in conjunction with $t(\varphi_f)$ to compute $\varphi = g(\varphi_f)$.

Algorithm 3.1

Now the LTL over finite trace to `C++` *monitor algorithm can be defined as follows:*

Input : *LTL_f formula φ_f.*

Output : *`C++` monitor class M.*

Step 1 : *From φ_f, generate the LTL formula φ as $\varphi = g(\varphi_f)$.*

Step 2 : *From φ, generate M that precisely rejects all the minimal bad prefixes of φ [14] using SPOT. Return M.*

3.2.2 Complexity

The complexity of LTL to `C++` monitor generation algorithm given in [14] is doubly exponential in the length of the input LTL formula. So Step 2 of Algorithm 3.1 is doubly exponential. But the worst case complexity is rarely experienced in practice. In Step 1, one extra propositional variable **alive** is added per temporal operator in the LTL formula. So Step 1 is linear. Hence the overall complexity is doubly exponential in the length of the input LTL_f formula.

3.2.3 Implementation

The Flow Monitor Generation Algorithm is implemented in our tool FlowMonGen. It automatically generates `C++`flow monitor classes from the given flow properties. FlowMonGen is written in `C++`. It takes a configuration file as input. The configuration file contains all the flow properties to verify and some relevant flow information about the flow model under verification. As output it produces one `C++`flow monitor class per flow property. All the monitor classes lie in two files: monitor.h and monitor.cc. It also generates a class called `local_flow_manager class`, which is responsible for instantiating and executing the flow monitor classes. The local_flow_manager

class lies in two files local_flow_manager.h and local_flow_manager.cc, generated by FlowMonGen. We discuss more about this class in Chapter 4.

A flow monitor class is a C++ encoding of a DFW. The transition function of the DFW is encoded as a member function *step()* in the monitor class. Each time the function *step()* is executed, one transition happens in the DFW. So executing one step of the monitor (DFW) means executing the monitor *step()* function once. In our case, all the states of a monitor (DFW) is accepting. A monitor rejects by funding no possible transition from a state.

3.3 Flow Monitoring Algorithm

Our goal is to verify that each flow f of flow type T satisfies each flow property P, associated with flow type T. Each such flow property P can be converted to a flow monitor class M_P. Now we define our Flow Monitoring Algorithm as follows:

Algorithm 3.2

For each flow f and associated flow monitor class M do the following:

Step 1 : *When f begins during the simulation, a monitor instance m_f of M is created and assigned to f.*

Step 2 : *While f is alive, execute the step() of the monitor m_f with atomic proposition* **alive** *= true.*

Step 3 : *If m_f rejects before f ends, record monitoring status = FAIL and delete m_f.*

Step 4 : *Else*

Step 4.1 : *When f ends, execute the step() of m_f with value of atomic proposition* ***alive*** *= false once.*

Step 4.2 : *If m_f rejects, record monitoring status = FAIL. Else record monitoring status = PASS.*

Step 4.3 : *Delete m_f.*

Flow Monitoring Algorithm is dynamic and decentralized. It is dynamic because all the flow monitor instances are created during the simulation synchronized with the beginnings of new flows. It is decentralized because there is one monitor instance per flow and flow property pair, as contrast to one monitor instance per trace property. For example, let us assume that there are two flow properties φ and ψ associated with flow type T. Let the flow monitor classes generated from flow properties φ and ψ be M_φ and M_ψ respectively. Also let us assume that during the simulation, three flows f_1, f_2, and f_3 are created of flow type T. Then there would be total $2 \times 3 = 6$ flow monitor instances, created during the simulation. Three of them are instances of M_φ and the other three are instances of M_ψ. Each of the three flows f_1, f_2, and f_3 has two monitor instances associated with them. One of type M_φ and another of type M_ψ.

3.4 Proof of correctness

To prove the correctness of the above two algorithms, we shall prove that the generated monitor M accepts a flow f iff its finite trace slice $\pi_f(0 \cdots n)$ finitely satisfies the formula φ_f.

Definition 3.5 A finite word x over alphabet Σ is a **good prefix** of language L iff for all infinite word $y \in \Sigma^\omega,\ x.y \in L$.

Lemma 3.1 ***alive*** *is the atomic proposition defined in Definition 3.4. Let* π *be an infinite trace that satisfies the property* (***alive*** U $G!$***alive***) *and* ***alive*** $\in \pi(n)$ *and* ***alive*** $\notin \pi(n+1)$*. Let* φ_f *be an LTL formula. Then the following two hold:*

1. *If the finite trace* $\pi(0 \cdots n) \models_f \varphi_f$*, then* $\pi(0 \cdots n+1)$ *is a good prefix of* $t(\varphi_f)$*.*

2. *If the finite trace* $\pi(0 \cdots n) \not\models_f \varphi_f$*, then* $\pi(0 \cdots n+1)$ *is a bad prefix of* $t(\varphi_f)$*.*

Proof 3.1 We prove it by Induction on φ_f.

Base Case : φ_f is an atomic proposition $p \neq a$.

- $\pi(0 \cdots n) \models_f p \ \Rightarrow\ p \in \pi(0) \ \Rightarrow\ \pi(0 \cdots n+1)$ is a good prefix for $t(p) = p$.
- $\pi(0 \cdots n) \not\models_f p \ \Rightarrow\ p \notin \pi(0) \ \Rightarrow\ \pi(0 \cdots n+1)$ is a bad prefix for $t(p) = p$.

Inductive Step :

1. $\varphi_f = \neg\psi$.
   - $\pi(0 \cdots n) \models_f \neg\psi \ \Rightarrow \pi\ (0 \cdots n) \not\models_f \psi \ \Rightarrow$ By IH $\pi(0 \cdots n+1)$ is a bad prefix of $t(\psi) \ \Rightarrow$ All infinite extension of $\pi(0 \cdots n+1) \not\models t(\psi) \ \Rightarrow$ All infinite extension of $\pi(0 \cdots n+1) \models \neg t(\psi) \ \Rightarrow\ \pi(0 \cdots n+1)$ is a good prefix for $\neg t(\psi) = t(\neg\psi) = t(\varphi_f)$.
   - $\pi(0 \cdots n) \not\models_f \neg\psi \ \Rightarrow\ \pi(0 \cdots n) \models_f \psi \ \Rightarrow$ By IH $\pi(0 \cdots n+1)$ is a good prefix of $t(\psi) \ \Rightarrow$ All infinite extension of $\pi(0 \cdots n+1) \models t(\psi) \ \Rightarrow$ All infinite extension of $\pi(0 \cdots n+1) \not\models \neg t(\psi) \ \Rightarrow\ \pi(0 \cdots n+1)$ is a bad prefix for $\neg t(\psi)$.
2. $\varphi_f = \varphi_1 \wedge \varphi_2$
   - $\pi(0 \cdots n) \models_f \varphi_1 \wedge \varphi_2 \ \Rightarrow\ \pi(0 \cdots n) \models_f \varphi_1$ and $\pi(0 \cdots n) \models_f \varphi_2 \ \Rightarrow$ By IH, $\pi(0 \cdots n+1)$ is a good prefix for both $t(\varphi_1)$ and $t(\varphi_2) \ \Rightarrow$ All

infinite extension of $\pi(0 \cdots n+1)$ satisfy both $t(\varphi_1)$ and $t(\varphi_2)$ $\Rightarrow$ All infinite extensions of $\pi(0 \cdots n+1) \models t(\varphi_1) \wedge t(\varphi_2)$ $\Rightarrow$ $\pi(0 \cdots n+1)$ is a good prefix for $t(\varphi_1) \wedge t(\varphi_2) = t(\varphi_1 \wedge \varphi_2) = t(\varphi_f)$.

- $\pi(0 \cdots n) \not\models_f \varphi_1 \wedge \varphi_2$ $\Rightarrow$ $\pi(0 \cdots n) \not\models_f \varphi_1$ or $\pi(0 \cdots n) \not\models_f \varphi_2$ $\Rightarrow$ By IH, $\pi(0 \cdots n+1)$ is a bad prefix for $t(\varphi_1)$ or $t(\varphi_2)$ $\Rightarrow$ All infinite extensions of $\pi(0 \cdots n+1)$ either does not satisfy $t(\varphi_1)$ or does not satisfy $t(\varphi_2)$ $\Rightarrow$ All infinite extensions of $\pi(0 \cdots n+1) \not\models t(\varphi_1) \wedge t(\varphi_2)$ $\Rightarrow$ $\pi(0 \cdots n+1)$ is a bad prefix for $t(\varphi_1) \wedge t(\varphi_2)$.

3. $\varphi_f = X\psi$

   - $\pi(0 \cdots n) \models_f X\psi$ $\Rightarrow$ $n \geq 1$ and $\pi(1 \cdots n) \models_f \psi$ $\Rightarrow$ **alive** $\in \pi(1)$ [Since $n \geq 1$] and by IH $\pi(1 \cdots n+1)$ is a good prefix of $t(\psi)$ $\Rightarrow$ All infinite extensions of $\pi(1 \cdots n+1)$ satisfy both **alive** and $t(\psi)$ $\Rightarrow$ All infinite extensions of $\pi(0 \cdots n+1) \models X(\textbf{alive} \wedge t(\psi))$ $\Rightarrow$ $\pi(0 \cdots n+1)$ is a good prefix of $X(\textbf{alive} \wedge t(\psi)) = t(X\psi) = t(\varphi_f)$
   - $\pi(0 \cdots n) \not\models_f X\psi$ $\Rightarrow$ Either $n = 0$ or $\pi(1 \cdots n) \not\models_f \psi$ $\Rightarrow$ Either **alive** $\notin \pi(1)$ [Since $n = 0$] or by IH $\pi(1 \cdots n+1)$ is a bad prefix of $t(\psi)$ $\Rightarrow$ Either all infinite extensions of $\pi(1 \cdots n+1) \not\models$ **alive** [Since **alive** $\notin \pi(1)$] or all infinite extensions of $\pi(1 \cdots n+1) \not\models t(\psi)$ $\Rightarrow$ All infinite extensions of $\pi(1 \cdots n+1) \not\models (\textbf{alive} \wedge t(\psi))$ $\Rightarrow$ All infinite extensions of $\pi(0 \cdots n+1) \not\models X(\textbf{alive} \wedge t(\psi))$ $\Rightarrow$ $\pi(0 \cdots n+1)$ is a bad prefix of $X(\textbf{alive} \wedge t(\psi))$.

4. $\varphi_f = \varphi_1 U \varphi_2$

   - $\pi(0 \cdots n) \models_f \varphi_1 U \varphi_2$ $\Rightarrow$ $\exists j \leq n$ such that $\pi(j \cdots n) \models_f \varphi_2$ and $\forall k \ 0 \leq k < j$, $\pi(k \cdots n) \models_f \varphi_1$ $\Rightarrow$ By IH $\exists j \leq n$ such that

$\pi(j \cdots n+1)$ is a good prefix of $t(\varphi_2)$ and $\forall k\ 0 \leq k < j,\ \pi(k \cdots n+1)$ is a good prefix of $t(\varphi_1)\ \Rightarrow\ \exists j \leq n$ such that $\pi(j \cdots n+1)$ is a good prefix of $t(\varphi_2) \wedge \mathbf{alive}$ [Since $j \leq n,\ \mathbf{alive} \in \pi(j)$] and $\forall k\ 0 \leq k < j,\ \pi(k \cdots n+1)$ is a good prefix of $t(\varphi_1)\ \Rightarrow\ \exists j \leq n$ such that all infinite extensions of $\pi(j \cdots n+1) \models t(\varphi_2) \wedge \mathbf{alive}$ and $\forall k\ 0 \leq k < j$, all infinite extensions of $\pi(k \cdots n) \models t(\varphi_1)\ \Rightarrow$ For all infinite extensions $\pi(n+2 \cdots)$, $\exists j \leq n$ such that $\pi(j \cdots n+1)\pi(n+2 \cdots) \models t(\varphi_2) \wedge \mathbf{alive}$ and $\forall k\ 0 \leq k < j,\ \pi(k \cdots n+1)\pi(n+2 \cdots) \models t(\varphi_1)\ \Rightarrow$ For all infinite extensions $\pi(n+2 \cdots),\ \pi(0 \cdots n+1)\pi(n+2 \cdots) \models t(\varphi_1) U (t(\varphi_2) \wedge \mathbf{alive})\ \Rightarrow\ \pi(0 \cdots n+1)$ is a good prefix for $t(\varphi_1) U (t(\varphi_2) \wedge \mathbf{alive}) = t(\varphi_1 U \varphi_2) = t(\varphi_f)$.

- $\pi(0 \cdots n) \not\models_f \varphi_1 U \varphi_2$. There are two cases:

  Case 1: $\forall i,\ 0 \leq i \leq n,\ \pi(i \cdots n) \not\models_f \varphi_2\ \Rightarrow$ By IH $\forall i\ 0 \leq i \leq n,\ \pi(i \cdots n+1)$ is a bad prefix of $t(\varphi_2)\ \Rightarrow\ \forall i\ 0 \leq i \leq n,\ \pi(i \cdots n+1)$ is a bad prefix of $t(\varphi_2) \wedge \mathbf{alive}\ \Rightarrow\ \pi(0 \cdots n+1)$ is a bad prefix of $t(\varphi_1) U (t(\varphi_2) \wedge \mathbf{alive})$

  Case 2: $\exists k \leq n$ such that $\pi(k \cdots n) \not\models_f \varphi_1$ and $\forall i,\ 0 \leq i \leq k,\ \pi(i \cdots n) \not\models \varphi_2\ \Rightarrow\ \exists k \leq n$ such that $\pi(k \cdots n+1)$ is a bad prefix of $t(\varphi_1)$ and $\forall i,\ 0 \leq i \leq k,\ \pi(i \cdots n+1)$ is a bad prefix of $t(\varphi_2)\ \Rightarrow\ \exists k \leq n$ such that $\pi(k \cdots n+1)$ is a bad prefix of $t(\varphi_1)$ and $\forall i,\ 0 \leq i \leq k,\ \pi(i \cdots n+1)$ is a bad prefix of $t(\varphi_2) \wedge \mathbf{alive}\ \Rightarrow\ \pi(0 \cdots n+1)$ is a bad prefix of $t(\varphi_1) U (t(\varphi_2) \wedge \mathbf{alive})$.

*Theorem 3.2 Let π be an infinite trace that satisfy the property (**alive** U G!**alive**)*

and ***alive*** $\in \pi(n)$ *and* ***alive*** $\notin \pi(n+1)$. *Let* φ_f *be an LTL formula. Let* M *be the minimal deterministic monitor generated from* φ_f *using Algorithm 1. The finite trace* $\pi(0 \cdots n) \not\models_f \varphi_f$ *iff* M *rejects* π *within* $n+1$ *steps.*

Proof 3.2 $\Rightarrow \pi(0 \cdots n) \not\models_f \varphi_f \Rightarrow \pi(0 \cdots n{+}1)$ is a bad prefix of $t(\varphi_f)$ [By Lemma 1] $\Rightarrow$ $\pi(0 \cdots n{+}1)$ is a bad prefix of $t(\varphi_f) \wedge (\textbf{alive}\ U\ G!\textbf{alive}) \Rightarrow$ either $\pi(0 \cdots n{+}1)$ is minimal or it has a minimal bad prefix $\pi(0 \cdots i)$ where $i \leq n \Rightarrow M$ rejects π within $n+1$ steps (By Theorem 3.1).

$\Leftarrow \pi(0 \cdots n) \models_f \varphi_f \Rightarrow \pi(0 \cdots n+1)$ is a good prefix of $t(\varphi_f)$ [By Lemma 1] $\Rightarrow \pi(0 \cdots n+1)$ is not a bad prefix for $t(\varphi_f) \wedge (\textbf{alive}\ U\ G!\textbf{alive})$ since by assumption $\pi(0 \cdots n+1)$ is not a bad prefix of $(\textbf{alive}\ U\ G!\textbf{alive}) \Rightarrow M$ does not reject $\pi(0 \cdots n+1)$.

Let M be the flow monitor class generated by FlowMonGen from a flow property P. Let m_f be the monitor instance of type M that is responsible to monitor the flow f. Algorithm 3.2 creates m_f with **alive** = true as soon as the flow begins and makes **alive** = false as soon as f ends. So definitely, the infinite trace whose prefix is the finite trace slice π_f of flow f satisfies the property (**alive** U $G!$**alive**). Theorem 3.2 guarantees that m_f rejects in Step 4.1 of Algorithm 3.2 iff π_f does not finitely satisfy P. So in Algorithm 3.2 the monitor instance m_f accepts a flow f iff flow f satisfies the flow property P.

The Flow Monitoring Framework presented in Chapter 4, implements this dynamic and decentralized Flow Monitoring Algorithm.

Chapter 4

Flow Monitoring Framework

Our goal is to automatically verify flow properties of a flow model using online monitoring technique. Each flow property of type T should be verified for each flow of type T. We already presented the algorithm for monitoring flows in Chapter 3. In this Chapter, we present our completely automated Flow Monitoring Framework that efficiently implements that algorithm enabling automatic flow monitoring.

4.1 Components

The complete Flow Monitoring Framework consists of three parts: (1) the Flow Library that provides the base classes and APIs, required to design flow models, (2) a flow model (written by the user), which is the model under verification (MUV), and (3) a set of flow-monitor classes (one flow monitor class per flow property) with the `local_flow_manager class`, generated by our FlowMonGen tool. The first component, the Flow Library, is the only component that never changes. The flow library is provided as part of our SystemC Flow Package, which can be downloaded at `http://sourceforge.net/projects/SystemCFlow/`. The second component is the `SystemC` model under verification and changes every time the user makes a modification to his model or wants to verify a different model. The third component depends on the flow properties to verify. Every time the user makes any change to the set of flow properties he/she wants to verify, the user regenerates the flow monitor classes

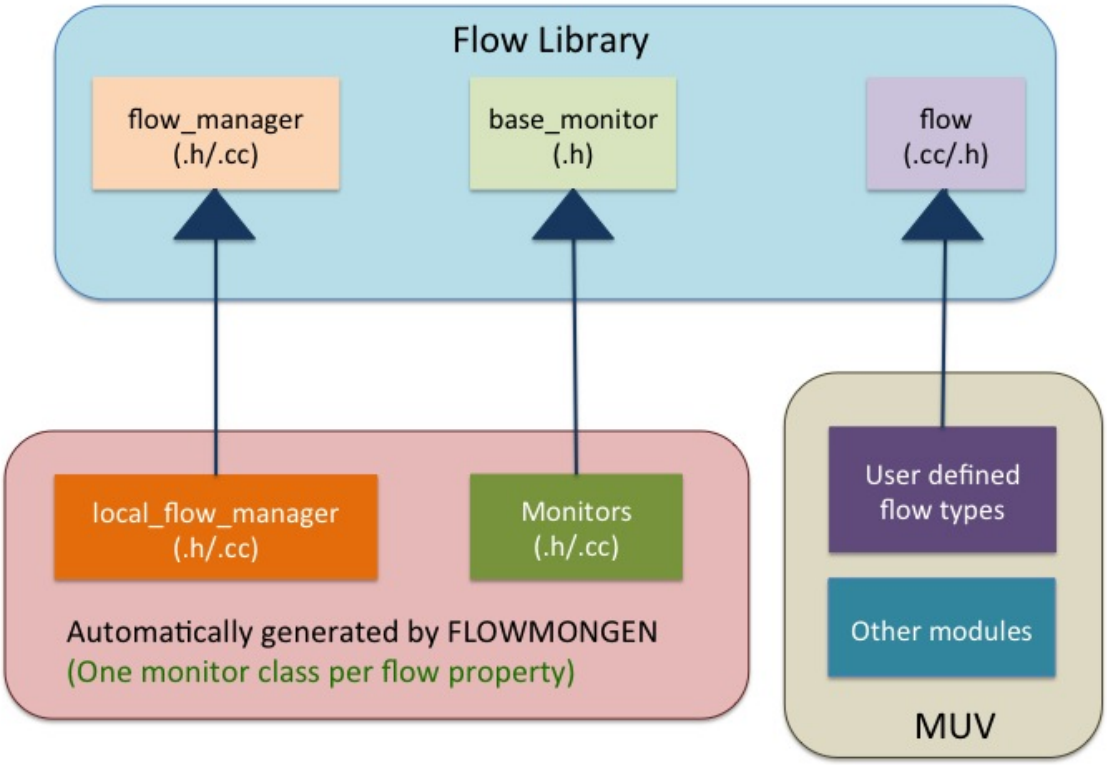

Figure 4.1 : The Flow Monitoring Framework

and the `local_flow_manager class` using the FlowMonGen tool, provided in our SystemC Flow Package. The `local_flow_manager class` needs to be regenerated every time a new property is added to the set of flow properties to verify. This is because it is the job of the `local_flow_manager class` to instantiate the flow monitor classes. So the `local_flow_manager` needs to know about all the flow monitor classes. Fig. 4.1 shows these three different components of the Flow Monitoring Framework. The blue arrows show `C++` inheritance relationship (will be discussed later in detail).

4.1.1 First Component: Flow Library

The first component, Flow Library, is described in detail in Chapter 2. In summary, it contains three classes: first, a `flow class` that is the base class of all the user-defined flow type classes in the MUV, second, a `flow_manager class` that has all the information about the flows that begin and end during a simulation, third, a `base_monitor class` that serves as a base class for all the flow monitor classes generated by FlowMonGen from the flow properties.

4.1.2 Second Component: MUV

The second component is the MUV, the flow model that the user intends to verify. The MUV can have one or more flow types. Each flow type is a class derived from the `flow class`. The MUV encodes how the flows travel through different `SystemC` processes using the methodology provided in Chapter 2.

4.1.3 Third Component: flow monitor classes and `local_flow_manager class`

The third component is the flow monitor classes and the `local_flow_manager class`. These are generated automatically from the flow properties using FlowMonGen tool. All the flow monitor classes are derived from the `base_monitor class`, provided in the Flow Library. This reduces the size of the `local_flow_manager class` because all the methods in the `local_flow_manager` class can use the common base class pointer base_monitor*.

The generated `local_flow_manager class` is derived from the `flow_manager class`. When a flow begins or ends the flow_manager class informs the local_flow_manager class by calling its APIs. The *alive_flows* database of `flow_manager class` is defined as a protected data member so that the local_flow_manager class can access the at-

tributes of any flow. The `local_flow_manager class` also has complete information about the flow monitor classes; which flow monitor class is associated with which flow type. So, `local_flow_manager class` is the one that dynamically creates the flow monitor instances, executes their step functions and deletes them when needed during the monitored simulation as described in Algorithm 3.2.

We have discussed about the first and second components in detail in Chapter 2. In this Chapter, we focus on the third component and how the three components work together to achieve the goal of flow monitoring.

4.2 flow monitors

A flow monitor class is a `C++` class generated from a flow property P using the FlowMonGen tool. Assume that P is a flow property of flow type T. P can refer to the flow attributes of T and the global variables in the MUV. The generated flow monitor class has one pointer member variable for each global variable, referred in P. Each such member variable is initialized in the monitor class's constructor to point to the corresponding global variable in the MUV. Each monitor class has two member functions: the constructor and a *step()*.

As mentioned earlier, each flow monitor is a `C++` encoding of a DFW (Deterministic Finite Word automaton). We use the `front_det_ifelse` encoding proposed in [14] to encode the transition function of that DFW. The flow monitor constructor is used to initialize the member variables of the flow monitor class to pint to the corresponding global variables of the MUV. It also sets the initial state of the DFW. The *step()* function encodes the transition function of the DFW and is defined as:

```
virtual bool step(flow* p, bool alive);
```

The *step()* function is virtual as it is first defined in the `base_monitor class` as an empty function and is overridden by all the flow monitor classes. The *step()* function takes two arguments: the pointer to the flow it is monitoring and a boolean variable called **alive**. The *step()* function returns a boolean value. The return value is true if the there is no possible transition from the current state and false otherwise.

When a flow f begins during the simulation, its corresponding monitor instance m is created. Since then, until f ends, m's *step()* is called one or more times with **alive** = true. If any of these steps rejects by returning true, the status of m is recorded as "FAIL" and m is deleted immediately. Else, when flow f ends, m's *step()* function is called immediately with **alive** = false and, depending on its return value, the status of m is recorded and m is deleted.(See Algorithm 3.2) Remember that for a flow f of type T, there is one such monitor instance m for every flow property P of type T. For example, if there are three flow properties of type T, there will be three different monitor instances, which will be monitoring only flow f.

4.3 local_flow_manager class

4.3.1 Simulation with and without monitors

The MUV can be run with monitors (monitored simulation) or without monitors (unmonitored simulation). In monitored simulation, the MUV needs to be modified to plug-in the calls to the monitors' *step()* function at the appropriate locations in it. This modification is called *instrumentation*.

Instrumentation can be done either manually or automatically. As the size of the MUV grows larger, manual instrumentation becomes cumbersome and error prone. Automatic instrumentation is hard since it needs a full-blown `C++` parser. Aspect-

Oriented Programming [35] is a good fit for this, since inserting monitor calls can be seen as cross-cutting concerns (aspects) in the MUV. But, unfortunately, the only aspect-oriented tool for `C++`, `AspectC++`, [36] does not support many join points necessary for monitoring. An example of such join point is Field Access join point, which denotes all the locations where a particular variable is assigned a value. This is an important join point in case of monitoring since it makes sense to run the monitors whenever some variable changes its value.

Since we do not have an automatic instrumentation tool and manual instrumentation is practically unusable, one of the major goals in this Flow Monitoring Framework is to support transition from unmonitored simulation to monitored simulation and vice versa with almost no instrumentation. Also, the unmonitored simulation should not put any monitoring overhead.

The derived class local_flow_manager overrides some virtual methods of its base class flow_manager to do some extra work only during monitored simulation. We want to build our framework in a way such that during unmonitored simulation the methods of the `flow_manager class` will be called and during monitored simulation the methods of the `local_flow_manager class` will be called. This should happen automatically. The only thing the user has to write is to indicate somewhere in the MUV if it is a monitored simulation or an unmonitored one.

This goal can be achieved best using `C++` polymorphism (base class pointer pointing derived class object). In case of monitored simulation, the global pointer *flow_manager* fmanager* should point to a local_flow_manager object (derived class object), whereas in an unmonitored simulation, *fmanager* points to a flow_manager object (base class object).

As we have seen in Chapter 2, a flow property can refer to global variables, more

precisely member variables of different `SystemC` modules. If a flow property refers to a member variable of some `SystemC` module M, the corresponding flow monitor needs access to the module object M. It is the job of local_flow_manager class to give each flow_monitor class access to its required `SystemC` module objects. For that, when the object of `local_flow_manager class` is constructed, the user needs to pass the addresses of all the `SystemC` modules referred in the flow properties to verify, as the argument to the constructor. The order of the parameters $M_1, \cdots, M_n$ is determined by the order, provided by the user in the input configuration file to the FlowMonGen tool, while generating the `local_flow_manager class` and the flow monitor classes. The Listing below shows the modification, needed to run the simulation of a flow model with and without monitor.

```
//For unmonitored simulation
flow_manager* manager = new flow_manager
  (<number of user-defined flow types>);


//For monitored simulation
/*Let M1,..,Mn be the global variables,
referred in the flow properties*/
flow_manager* fmanager =
  new local_flow_manager
  (<number of user-defined flow types>,
  &M1, ... ,&Mn );
```

Thus to execute the simulation with and without monitors, the user has to change only one line of code in the entire MUV.

4.3.2 Dynamic creation and deletion of flow monitor instances

During the simulation, a `SystemC` process may begin a flow f by calling the *begin_flow* function of `flow_manager class`. After doing all the book-keepings and security checks (as described in Chapter 2), the `flow_manager class` calls the function *local_begin_flow(unsigned int id, int type)*, where *id* and *type* are the *flow_id* and flow type id of f respectively. This function is implemented as an empty virtual function in `flow_manager class` and overridden by `local_flow_manager class`.

In a monitored execution, when a flow f of flow type T begins, the process first calls $begin\_flow$ of `flow_manager class`, which calls the $local\_begin\_flow$ of `local_flow_manager class`. The $local\_begin\_flow()$ creates one instance of each flow monitor class of type T. All these monitor instances are responsible to monitor only flow f. We say that these monitor instances are 'assigned to' f. After the monitor instances are created, the `local_flow_manager class` executes one step of each monitor instance and returns.

During the simulation, a `SystemC` process may end a flow f by calling the end_flow function of `flow_manager class`. $end\_flow()$ does some book keeping, error checking and then it deletes f if no other process is using f. Before deleting f, it calls a function *local_end_flow(unsigned int id, int type)*. This is defined as an empty function in `flow_manager class` and `local_flow_manager class` overrides this to do some extra work during monitored simulation. For each flow monitor instance m assigned to flow f, *local_flow_manager::end_flow()* does the followings:

1. Execute one step of m with **alive** = true.

2. If *step()* returns true, record status of m = FAIL and go to Step 4. Else execute one step with **alive** = false.

3. If *step()* returns false, record status of m = PASS. Else record status of m = FAIL.

4. Delete m.

Thus `local_flow_manager class` takes care of dynamically creating and deleting monitor instances synchronizing with beginning and ending of flows.

4.3.3 Executing steps of a flow monitor

The Flow Monitoring Framework provides two APIs to the user to execute a step of a flow monitor from a location in the MUV. The first is:

```
fmanager->monitor_flow(unsigned int flow_id);
```

This function does not do anything in non-monitored execution. But in the monitored execution, it executes one step of all the flow monitor instances, associated with flow f. The second one is:

```
fmanager->monitor_all();
```

This function does not do anything in non-monitored execution. In the monitored execution, it executes one step of all flow monitor instances.

4.4 Sampling

Now the question is when to execute the steps of the flow monitors. In monitoring, a step of a monitor is executed by *sampling* the state of the MUV. During a monitored simulation, one can only sample the state of the MUV for a finite number of times. So it is really important to decide when to sample to obtain the maximum power of monitoring. Too little sampling might miss some important state of the system.

Again, too frequent sampling might be too expensive in terms of runtime and sometimes redundant. In case of flow monitoring, the naive approach is to sample after every statement of the MUV and at each sampling point, execute one step of all the flow monitor instances of all the live flows. But there are two main drawbacks of this approach: inefficiency and redundancy. At any point during the simulation, there can be a large number of flows alive in the system, each having multiple flow monitor instances associated with it. If we execute all the flow monitor instances at each sampling point, it will incur a large monitoring overhead on the simulation time of the MUV. On the other hand, not all flows change their states at every sampling point. A flow changes its state only when any of its flow attributes is written. So it is redundant to monitor every flow at every sampling point.

There are three main goals we would like achieve in the sampling mechanism of our Flow Monitoring Framework. First is 100% coverage. This means that the *step()* of a flow monitor should be executed frequently enough to capture every change in the state of the MUV relevant to that flow monitor. Second is 0% redundancy. This means that we should execute the step of a monitor only when some flow attribute of the flow, it is monitoring, or some global variable, it refers to, changes its value. Third, sampling should require no manual instrumentation of the MUV.

Using the Flow Monitoring Framework, user can sample at different resolution without any instrumentation of the MUV. In this subsection, we discuss how to sample at different resolution automatically during the monitored simulation of a flow model.

4.4.1 Sampling at value change of a flow attribute

Let us assume that the MUV has a flow type class T that has a flow attribute *int a*. Let f be a flow of type T. The Flow Monitoring Framework can automatically execute all flow monitor instances associated with flow f, every time the flow attribute a of flow f is written. For that the setter function of flow attribute a has to be defined in flow type class T as follows:

```
void set_a(int val) {set_attribute<int>(a, val);}
```

The function *set_attribute()* is defined in the `flow class` of Flow Library as a template function as follows:

```
template <typename T>
void set_attribute(T& att, const T& val);
```

The template type is the type of the flow attribute (in this case, integer). In case of non-monitored simulation, this function just sets the value of the attribute. In case of monitored simulation, after setting the value, it executes one step of all flow monitor instances associated with the flow, whose attribute's value has been set.

4.4.2 Sampling at value change of a global variable

Sometimes flow properties can refer to the global variables of the MUV. In `SystemC`, the global variables are mostly member variables of `SystemC` modules. The user might want to execute the flow monitors whenever that global variable is written a value. That can be done by adding a call to *flow_manager::monitor_all()* function at the end of the the setter method of that global variable:

```
//In the beginning of the file
#include "flow_manager.h"
```

```
extern flow_manager* fmanager;
...
//Let G be a global variable of type integer
void set_G(int val){
        G = val;
        fmanager->monitor_all();
}
...
```

In a non-monitored execution, this does nothing. But in the monitor execution, this executes all the flow monitors associated with all live flows. I would like to emphasize here that this is part of designing the flow model, not instrumentation.

4.4.3 Sampling at `SystemC` kernel phases

Sometimes the user might not want to capture all the changes in the system state. For that our Flow Monitoring Framework also provides a way to sample at lower granularity, for example, whenever a thread process suspends or whenever a delta cycle ends during the simulation. In [37], Tabakov et al. defines 18 `SystemC` kernel phases. In our Flow Monitoring Framework, the user can associate a set S of `SystemC` kernel phases with any flow property P. Each time any kernel phase in S occurs, the flow monitor instances of P execute one step automatically. The list of kernel phases supported by our Flow Monitoring Framework is given in Table 4.1

Our tool FlowMonGen takes a flow property associated with a set of kernel phase macros (given in the table above) as input. Each flow property can

Kernel Phase Name	Sampling location
MON_INIT_PHASE_BEGIN	Before initialization phase begins
MON_INIT_UPDATE_PHASE_BEGIN	Before initialization update phase begins
MON_INIT_UPDATE_PHASE_END	After initialization update phase ends
MON_INIT_DELTA_NOTIFY_PHASE_BEGIN	Before initialization delta notification phase begins
MON_INIT_DELTA_NOTIFY_PHASE_ENDS	After initialization delta notification phase ends
MON_INIT_PHASE_END	After initialization phase ends
MON_DELTA_CYCLE_BEGIN	Before a delta cycle begins
MON_DELTA_CYCLE_END	After a delta cycle ends
MON_EVALUATE_PHASE_BEGIN	Before an evaluation phase begins
MON_EVALUATE_PHASE_END	After an evaluation phase ends
MON_UPDATE_PHASE_BEGIN	Before an update phase begins
MON_UPDATE_PHASE_END	After an update phase ends
MON_DELTA_NOTIFY_PHASE_BEGIN	Before a delta notification phase begins
MON_DELTA_NOITIFY_PHASE_END	After a delta notification phase ends
MON_TIMED_NOTIFY_PHASE_BEGIN	Before a timed notification phase begins
MON_TIMED_NOTIFY_PHASE_END	After a timed notification phase ends
MON_METHOD_SUSPEND	After an sc_method has ended execution
MON_THREAD_SUSPEND	After an sc_thread has suspended

Table 4.1 : Kernel phase sampling points

have its own set of kernel phases, it is sensitive to. As output FlowMonGen generates the `local_flow_manager class` and the flow monitor classes such that `local_flow_manager class` automatically executes all the flow monitor instances associated with flow property P, whenever any kernel phase k, P is sensitive to, occurs.

When a kernel phase K occurs, the `local_flow_manager class` executes all the flow monitors sensitive to hernel pause K. For that, the `local_flow_manager class` has to know exactly when K occurs during the simulation. In the original `SystemC` package, only the `SystemC` kernel knows this information. To expose this information to our `local_flow_manager class`, we have put a minimal patch on the `SystemC`-2.3 kernel, similar to the one proposed in [10], but smaller in size. Our patch only contains 85 lines of code and is easily portable to the future `SystemC` releases.

Chapter 5

Experimental Evaluation

This section presents a case study, where we designed a flow model using our Flow Library and verified some flow properties of this model using our Flow Monitoring Framework. We ran a set of flow monitoring experiments with this flow model. We analyzed the result to see how the runtime overhead due to monitoring varies with different aspects of the model such as sampling rate, maximum number of live flows, ratio of user clock frequency to system clock frequency, total number of flows etc.

5.1 Airline Reservation System: Model under verification

The flow model we have designed, implements a multi-user, concurrent system for reserving and purchasing airline tickets. It has approximately 3800 lines of code. This model has one flow type: 'request' to book a trip. The users of the system submit requests to reserve plane tickets by specifying the starting and ending airports of the trip, the date of travel (if it is a return-trip request, there should be two dates for going and coming back) and a few other input attributes. The system uses a randomly generated flight database to find a direct flight or a sequence of up to three connecting flights. Those are returned back to the users as output attributes. The entire set of attributes in the corresponding flow type class 'request' is given below:

```
class request : public flow{
//Constructor and Functions
```

```
...
//Flow attributes
private:
        //Input Flow Attributes
        int source;       // Starting point of the trip
        int destination; // Ending point
        int date;         // Travel date
        int date2;        // Return date
        int seats;        // Number of passengers
        std::string category;    //business class, economy class etc
        bool returnflight; //True if this is a return trip request
        //Output flow attributes
        bool status; //False if not enough resource to process request
        list_t* connecting_flights; //Connecting flights for going
        list_t* connecting_flights2; //Connecting flights for coming
        //Other attribute
        //True if the request is in speculative state (explained later
        bool is_speculative;
}
```

The Airline Reservation System model contains four modules connected by finite-capacity channels. These modules are: user, IO, master and planner. There are two clocks defined in this model: user clock and system clock. The user module operates on the user clock and the other three modules operate according to the system clock. The user module simulates the users of this model and submits a new request at every clock cycle of the user clock with different values of input attributes. The other three

modules process the requests to generate outputs and send them back to the user module.

Each module has one or more thread processes in it. A request travels through those processes to get processed. If two processes belong to the same module, a request is transferred from one process to another using some bounded queue, defined locally inside that module. If two processes belong to two different modules, a request is transferred from one process to another using some finite capacity channel, connecting those two modules. The sum of the capacities of all these bounded queues and bounded channels puts a bound on the maximum number of live requests, flowing through the system during any simulation. The modules use events to synchronize reading and writing to the queues and the channels. This is a model of a reactive system that is intended to run forever.

Fig 5.1 shows the possible paths taken by a request in our model. A request is generated by a thread in the user module. Then it travels to a thread in IO module. IO module checks if it has space in its internal queue to store this request as a pending request. It not, it sets the status of the request to false and sends the request back to the users module that ends the request. Else the IO module sends the request to the master module after setting its status to true. If it is a 1-way trip request, master module just sends it to the planner module. If it is a return trip request, the master module sets the speculative_bit of the request to true and sends the request to the planner module to process trip1. It also keeps a copy of the same request in its internal queue to send that again to the planner in the near future to process trip2. Upon receiving a request from master module, the planner module checks its speculative_bit. If the speculative_bit is false, then the planner module checks if it is a return trip request. If it is not a return trip request, the planer module sets

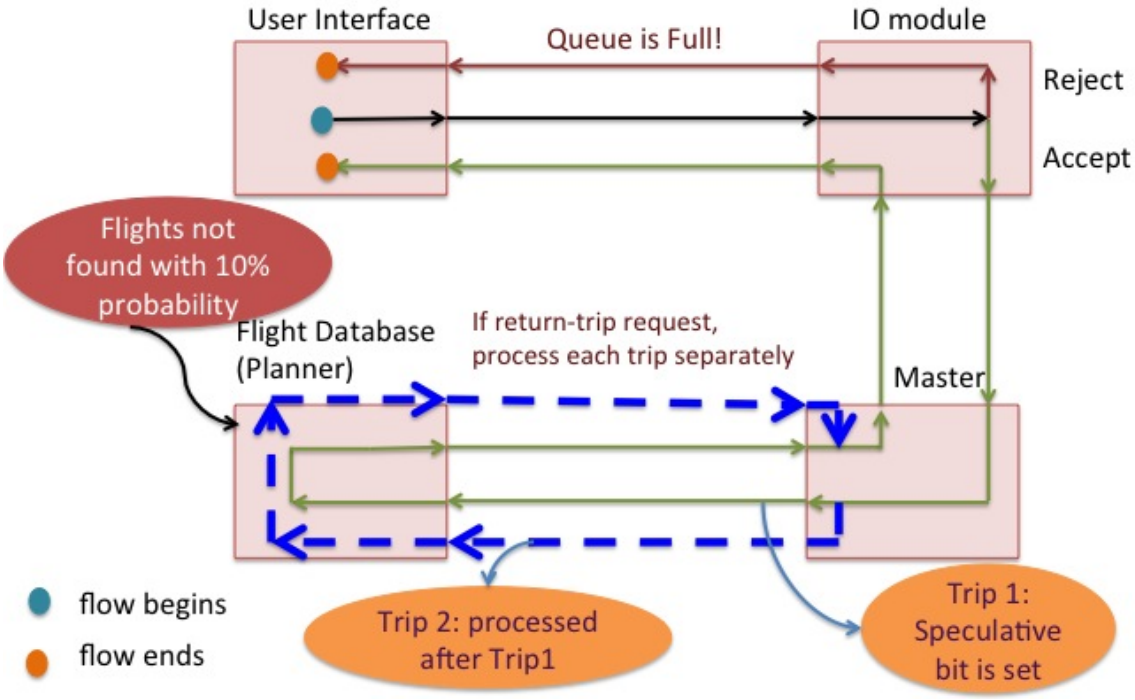

Figure 5.1 : A 'request' to book a trip (1-way or return trip) travels through the modules of the Airline Reservation System model.

the flow attribute connecting_flights to some randomly generated flights and send the 1-way request back to the master module. If it is a return-trip request, the planner module sets the connecting_flights to some randomly generated connecting flights and speculates the connecting_flights2 as the reverse of connecting_flights. Here we introduce an option of not finding connecting flights with a probability of 10%. Now the planner module sends the request back to the master. If the speculative_bit of the received request is true, the planner module sets its connecting_flights2 with some randomly generated connecting flights and sets the speculative_bit to false. Now the planner module sends the request back to the master module. Upon receiving a request from the planner module, the master module first checks its speculative_bit. If the speculative_bit is true, the master module just discards this request. Else, the master module sends this request to the IO module. IO module sends it to the users module that ends the flow.

5.2 Flow properties of Airline Reservation System

We have verified three flow properties of our Airline Reservation System model using our Flow Monitoring Framework.

5.2.1 Property 1

The first property is a liveness property that asserts that every request is eventually processed by the Airline Reservation System. It says that eventually globally the attribute connecting_flights should be non-empty and if it is a return trip request, then eventually globally the attribute connecting_flights2 should be nonempty too. If the user submits requests more frequently than the system can process (user clock frequency is greater than system clock frequency), some requests will be rejected by

the IO module and will be sent back to the user module without being assigned flights by the planner module. These requests will not satisfy this property. But if the user clock frequency is less than or equal to the system clock frequency, all the requests get processed, though some requests do not find flights with a probability of 10% at planner module. So when the user clock frequency is greater than the system clock frequency, the failure rate increases with the ratio of user clock frequency to the system clock frequency. The first property is written as the input to FlowMonGen tool as:

```
request* r
F (G (!"(r->get_connecting_flights())->empty()")) &&
("r->is_return()" -> F(G(!"(r->get_connecting_flights2())->empty()")))
```

5.2.2 Property 2

The second property captures the behavior of speculative bit for a return-trip request. For each return-trip request, eventually the speculative bit should be set to true at master module and then eventually globally it should be set to false by planner module. Similar to Property 1, if the user clock frequency is greater than the system clock frequency, some requests will not reach the master module and will not satisfy this property. This property is written as the input to FlowMonGen tool as:

```
request* q
"q->is_return()" -> F("q->get_speculative_status()"
&& FG(!"q->get_speculative_status()"))
```

5.2.3 Property 3

The third property is a safety property that makes sure that all the connective flights have legs less than or equal to the global variable "max_legs", defined in the planner module. This property shows how to refer to a global variable in a flow property. Property 3 is written as the input to FlowMonGen tool as:

```
request* p
G "((p->get_connecting_flights())->size())
<= (planner->get_max_legs())"
```

5.3 Experimental setup

In monitoring the main concern is the overhead that the monitors put on the runtime of the model. The runtime overhead of flow monitoring may depend on multiple factors, such as the ratio of the frequency of the user clock to the frequency of the system clock (let us call it *frequency_ratio*), the maximum number of flows that can be live in the system at the same time (let us call it *max_live_flows*), the sampling rate of the monitor (let us call it *sampling_rate*), the total simulation time (let us call it *simulation_time*), and the total number of flows that begin during the simulation (let us call it *num_flows*). It is obvious that the runtime overhead increases with the num_flows and the simulation_time because the number of flow monitor instances is proportional to num_flows, which increases with the simulation_time.

So the main goal is to find out how the runtime overhead is affected by the first three factors: frequency_ratio, max_live_flows and sampling_rate. For that we have setup the following experiments for each of the above three properties. We fix simulation_time to $100,000$ SC_NS and the user clock cycle time to 10 SC_NS. So the

total number of flows that begin during the simulation is fixed to $(100,000/10) = 10,000$. The maximum number of alive flows in the Airline Reservation System model is equal to (total capacity of all the bounded queues + total capacity of all the bounded channels) + (the number of processes that operate on requests, where each process can store at most one flow). For simplicity of experimental setup, we make all the bounded queues and channels of equal capacity (let us call it C). There are 4 bounded channels and 8 bounded queues in our model. The total number of processes that operates on flows is 11. So in Airline Reservation System model, $max\_live\_flows = 12 \times C + 11$. Now we want to see how the runtime overhead changes with C.

For experimental purpose, we consider 10 values of C (capacity): 100 to 1000, separated by 100; 20 values for frequency_ratio (user clock frequency/system clock frequency): 0.1 to 2, separated by 0.1; and 3 values for sampling_rate: sampling at value change of flow attributes, sampling at delta cycle end and sampling at thread suspensions. This leaves us with $10 \times 20 \times 3 = 600$ combinations for each property. For each combination we ran the monitored simulation for 10 times and took the average of the runtimes.

To measure the overhead in runtime due to monitoring, we consider the runtime of non-monitored execution of Airline Reservation System model as our baseline. In that case, there is no sampling points. But there are still 10 values for C and 20 values for frequency_ratio. This gives as $10 \times 20 = 200$ combinations for running without monitors. For each combination, we ran 10 times and took the average runtime. We calculated the monitoring overhead as a percentage increase of the corresponding (equal values for C and frequency_ratio) baseline runtime.

5.4 Experimental result and analysis

We have some interesting observations from our experiments. One interesting observation is that one can stress-test the behavior of the system by increasing the user clock frequency beyond the system clock frequency and see how different flows behave. Also one can observe the system's behavior at different values for the input attributes. Many times in a flow model, the path taken by a flow depends on the values of the input attributes (Example: ATM server).

Regarding the runtime overhead, there are some important findings about its variation with respect to frequency_ratio, max_live_flows (proportional to C) and sampling_rate. Similar results have been found for all the three flow properties.

5.4.1 Sampling at value change of flow attributes

When sampling_rate is at value change of flow attributes, the runtime overhead does not change much with the increase of max_live_flows or frequency_ratio. This is justified by the fact that in case of sampling at value change, we only execute the monitors of a flow when any of its attributes is assigned a value. So total number of monitor calls during the entire simulation is equal to the total number of flows multiplied by attribute assignments for each flow (constant for a given model). So the number of monitor calls only depends on the total number of flows begun during the simulation and nothing else.

5.4.2 Sampling at delta cycle end

In case of sampling at delta cycle end, the monitors of all live flows are executed at each sampling points. So as max_live_flows increases, the number of queued requests increases and thus total number of calls to the monitors increases. Thus when there

are pending requests (user clock frequency is greater than system clock frequency), the monitoring overhead increases with max_live_flows or C. Similarly, as the frequency_ratio increases, the system queues more and more pending requests and the monitoring overhead increases.

5.4.3 Sampling at thread suspensions

In case of sampling at thread suspensions, at each sampling point, the monitors of all the live flows are executed. So the variation in monitoring overhead is very similar to the case of sampling at delta cycle ends. The overall overhead is slightly higher than the overhead at sampling at delta cycle end.

Below we present the runtime overhead for Property 1. The runtime overheads of Property 2 and 3 show similar variations. Fig 5.2 shows the percentage runtime overhead per flow monitor with respect to the ratio of user clock and system clock frequency. Each point in this graph is the average of 100 runtime overhead values, 10 for each value of C from 100 to 1000. The percentage runtime overhead per flow monitor is calculated as (((Runtime of the monitored simulation - Runtime of corresponding non-monitored simulation)/Runtime of corresponding non-monitored simulation) * 100%) / Number of monitors. The number of monitors is equal to the total number of flows begun, which is 10,000 in our experiments. Similarly Fig 5.3 shows the percentage runtime overhead per monitor with respect to the maximum number of live flows. Each point in this graph is the average of 200 runtime overhead values, 10 for each value of frequency_ratio from 0.1 to 2. Both the graphs shows that overhead of sampling at value changes at flow attributes is negligible and sampling at kernel phases is far more expensive than sampling at value changes of the flow attributes.

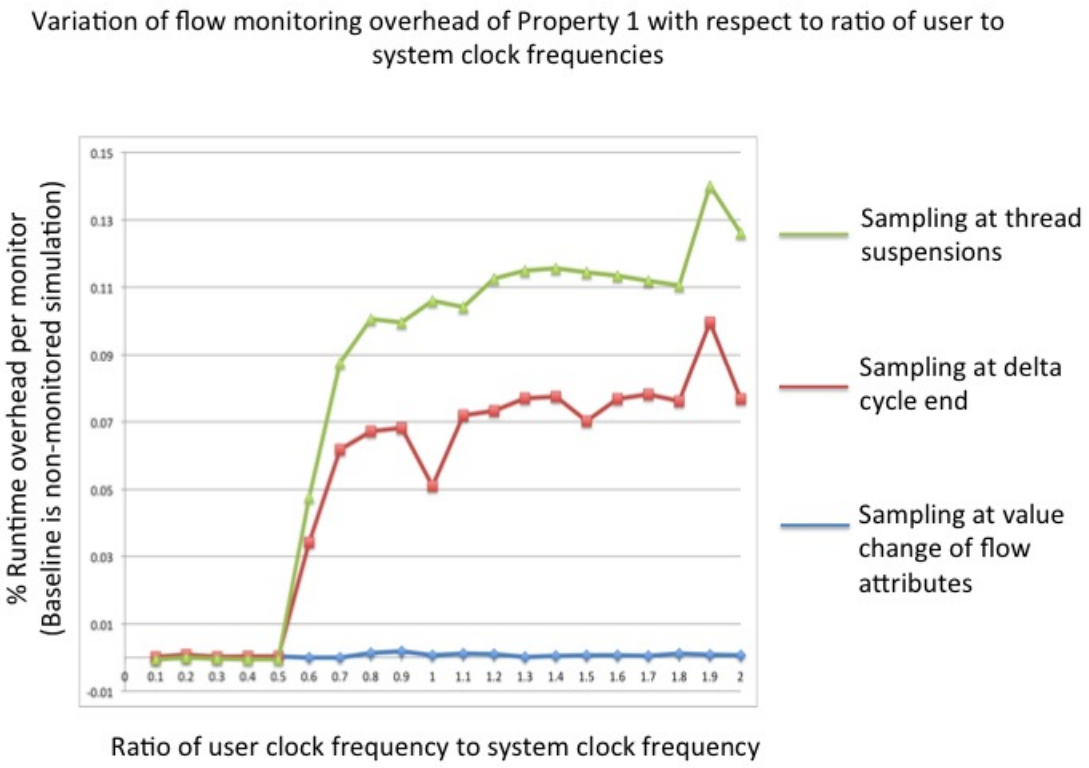

Figure 5.2 : Percentage runtime overhead of flow monitoring of Property 1 with respect to ratio of user and system clock frequencies. The three curves shows the runtime overhead for three different sampling rates.

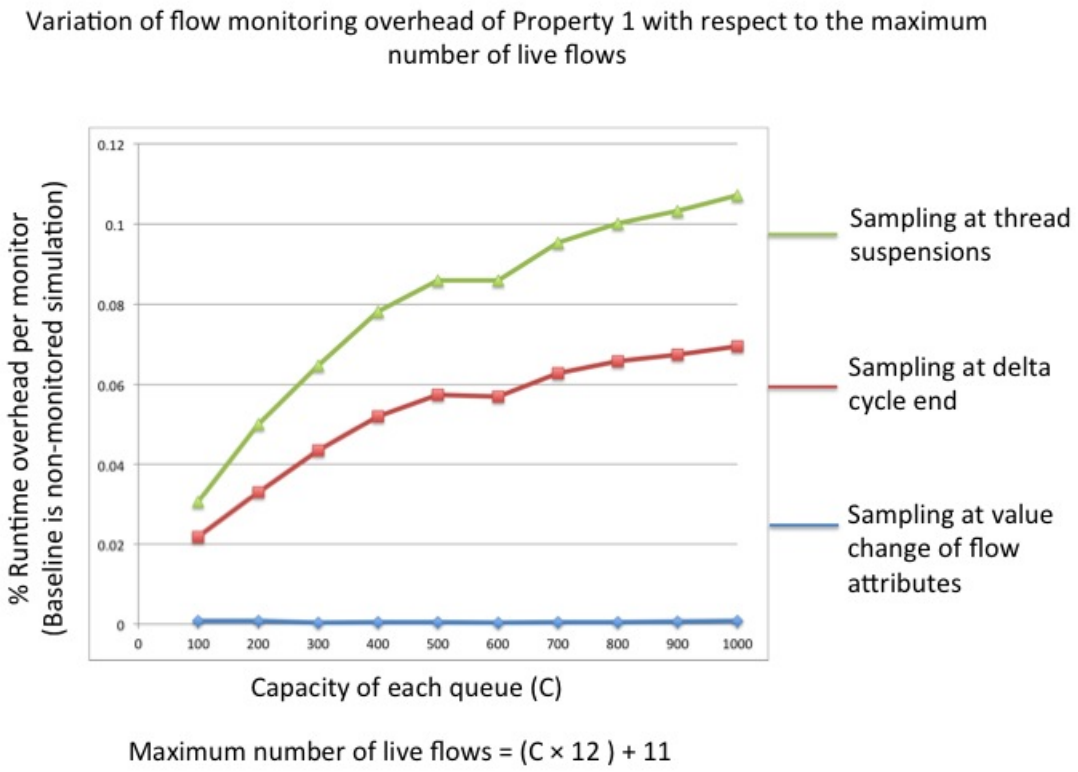

Figure 5.3 : Percentage runtime overhead of flow monitoring of Property 1 with respect to maximum number of alive flows. The three curves shows the runtime overhead for three different sampling rates.

Chapter 6

Concluding Remarks

In this section, we summarize the contribution of this thesis while indicating directions for future work.

6.1 Summary of contribution

The main contribution of this thesis is four fold. The first is the introduction of the concept of flows in the context of `SystemC` and methodology to implement flows in `SystemC` models. Though there have been prior work on modeling the formalism of flows in different fields of computer science, this is the first time to implement it in `SystemC`. Using our light-weight Flow Library, one can design a flow model with multiple flow types indicating the different types of jobs that system can perform. Flow Library provides the base class to model different flow types and their flow attributes in a flow model. It also provides APIs using which a `SystemC` process can begin a flow, end a flow, transfer a flow to another process. This thesis also shows how the user can implement branching and merging of a flow using the Flow Library. Also there is an efficient error-checking built-in the Flow Library to prevent mishandling of flow objects by the flow model implementation.

The second one is our LTL_f to `C++`flow monitor generation algorithm that is implemented in our FlowMonGen (Flow Monitor Generation) tool. FlowMonGen generates one `C++` monitor class per flow property. Though the worst case complexity

is double exponential, prior work [14] shows that LTL to `C++` monitor generation works well in practice and we have shown in Chapter 3 that LTL to LTL_f conversion is linear.

The third contribution is the dynamic and decentralized Flow Monitoring Algorithm. This creates and deletes monitor instances dynamically during the simulation synchronizing with beginning and ending of flows. This algorithm is decentralized because instead of generating one monitor instance per flow property, it generates one monitor instance per flow and flow property pair.

The fourth contribution of this thesis is a completely automated flow monitoring framework for SystemC. Using this framework, one can monitor one or more flow properties about different flow types of the flow model under verification. This framework allows automatic sampling for monitoring at different abstraction levels such as value change of flow attributes, ending of delta cycles, suspensions of thread processes etc.

The Flow Library and the FlowMonGen tool are available as the parts of a package, called SystemC Flow Package, available at: `https://sourceforge.net/projects/SystemCFlow`.

6.2 Future Work

This work leads to two other interesting works: hierarchy among flows and online monitoring of inter-flow properties.

6.2.1 Hierarchy of flows

A flow performs a job. It consists of multiple steps. Sometimes a flow can do similar work again and again. For example, suppose a system multiplies two numbers by

repetitive additions. If multiplication and addition are two flow types, it is easy to see that a flow f of type multiplication consists of many flows of type addition. Those addition flows can be thought as sub-flows of the multiplication flow f. The flow type addition is a sub-flow type of flow type multiplication. A flow can be composed of multiple types of sub-flows. So, a flow type can have more than one sub-flow types. Similarly a flow type can be sub-flow type of multiple flow types. For example, the same system can also use addition to compute square of a number. In that case, flow type addition is sub-flow type of both multiplication and square flow types. Another real-life example can be processing an image using divide and conquer method. First, divide the image into smaller segments. Process each segment separately and then combine the result to produce the output. The processing of the whole image is a parent flow and processing each segment is a sub-flow of that parent flow. It will be good to upgrade the whole SystemC Flow Package to support hierarchy of flows.

6.2.2 Monitoring inter-flow properties

There can be interesting properties about how different flows interact with each other. Some of these interactions happen using global variables. Since the intra-flow properties can refer to any global variables, some interactions among flows can be captured in intra-flow proper ties. But there can be interactions that cannot be captured using intra-flow properties, that can refer to the attributes of only one flow. For example, let us assume the flow property that says a flow outputs SUCCESS if and only if all its sub-flows output SUCCESS. This property can only be written as an inter-flow property that can refer to the attributes of more than one flow. It will be interesting to extend this Flow Monitoring Framework to monitor inter-flow properties as well.

Bibliography

[1] C. Helmstetter, F. Maraninchi, L. Maillet-Contoz, and M. Moy, "Automatic generation of schedulings for improving the test coverage of systems-on-a-chip," in *Formal Methods in Computer Aided Design, 2006. FMCAD'06*, pp. 171–178, IEEE, 2006.

[2] S. Liao, G. Martin, S. Swan, and T. Grötker, *System design with SystemC*. Kluwer Academic Pub, 2002.

[3] L. Charest *et al.*, "A vhdl/systemc comparison in handling design reuse," in *System-on-Chip for Real-Time Applications*, pp. 41–50, Springer, 2003.

[4] D. C. Black, *SystemC: From the ground up*, vol. 71. Springer, 2010.

[5] A. Hoffman, T. Kogel, and H. Meyr, "A framework for fast hardware-software co-simulation," in *Proceedings of the conference on Design, automation and test in Europe*, pp. 760–765, IEEE Press, 2001.

[6] M. Leucker and C. Schallhart, "A brief account of runtime verification," *Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009.

[7] F. Chen and G. Roşu, "Mop: an efficient and generic runtime verification framework," in *ACM SIGPLAN Notices*, vol. 42, pp. 569–588, ACM, 2007.

[8] H. Foster, "Assertion-based verification: Industry myths to realities (invited tutorial)," in *Computer Aided Verification*, pp. 5–10, Springer, 2008.

[9] H. D. Foster, A. C. Krolnik, and D. J. Lacey, *Assertion-based design*. Springer, 2004.

[10] D. Tabakov and M. Y. Vardi, "Monitoring temporal systemc properties," in *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pp. 123–132, IEEE, 2010.

[11] M. Y. Vardi, "Formal techniques for systemc verification; position paper," in *Design Automation Conference, 2007. DAC'07. 44th ACM/IEEE*, pp. 188–192, IEEE, 2007.

[12] L. Pierre and L. Ferro, "A tractable and fast method for monitoring systemc tlm specifications," *Computers, IEEE Transactions on*, vol. 57, no. 10, pp. 1346–1356, 2008.

[13] A. Habibi, A. Gawanmeh, and S. Tahar, "Assertion based verification of psl for systemc designs," in *System-on-Chip, 2004. Proceedings. 2004 International Symposium on*, pp. 177–180, IEEE, 2004.

[14] D. Tabakov, K. Y. Rozier, and M. Y. Vardi, "Optimized temporal monitors for systemc," *Formal Methods in System Design*, vol. 41, no. 3, pp. 236–268, 2012.

[15] D. Tabakov and M. Y. Vardi, "Automatic aspectization of systemc," in *Proceedings of the 2012 workshop on Modularity in Systems Software*, pp. 9–14, ACM, 2012.

[16] S. Dutta, D. Tabakov, and M. Y. Vardi, "Chimp: a tool for assertion-based dynamic verification of systemc models," 2012.

[17] A. Pnueli, "The temporal logic of programs," in *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pp. 46–57, IEEE, 1977.

[18] D. Schwartz-Narbonne, C. Chan, Y. Mahajan, and S. Malik, "Supporting rtl flow compatibility in a microarchitecture-level design framework," in *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pp. 343–352, ACM, 2009.

[19] D. Sethi, Y. Mahajan, and S. Malik, "Specification and encoding of transaction interaction properties," *Formal Methods in System Design*, vol. 39, no. 2, pp. 144–164, 2011.

[20] Y. Mahajan and S. Malik, "Automating hazard checking in transaction-level microarchitecture models," in *Formal Methods in Computer Aided Design, 2007. FMCAD'07*, pp. 62–65, IEEE, 2007.

[21] Y. Mahajan, C. Chan, A. Bayazit, S. Malik, and W. Qin, "Verification driven formal architecture and microarchitecture modeling," in *Formal Methods and Models for Codesign, 2007. MEMOCODE 2007. 5th IEEE/ACM International Conference on*, pp. 123–132, IEEE, 2007.

[22] W. Damm and D. Harel, "Lscs: Breathing life into message sequence charts," *Formal Methods in System Design*, vol. 19, no. 1, pp. 45–80, 2001.

[23] S. Maoz, D. Harel, and A. Kleinbort, "A compiler for multimodal scenarios: Transforming lscs into aspectj," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 4, p. 18, 2011.

[24] D. Harel, A. Marron, and G. Weiss, "Behavioral programming," *Communications of the ACM*, vol. 55, no. 7, pp. 90–100, 2012.

[25] M. Talupur and M. R. Tuttle, "Going with the flow: Parameterized verification using message flows," in *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, p. 10, IEEE Press, 2008.

[26] J. O'Leary, M. Talupur, and M. R. Tuttle, "Protocol verification using flows: An industrial experience," in *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, pp. 172–179, IEEE, 2009.

[27] F. Leymann and D. Roller, *Production workflow: concepts and techniques*. Prentice Hall PTR Upper Saddle River, 2000.

[28] M. Weske and G. Vossen, "Workflow languages," in *Handbook on Architectures of Information Systems*, pp. 359–379, Springer, 1998.

[29] S. Abiteboul, P. Bourhis, and V. Vianu, "Comparing workflow specification languages: a matter of views," *ACM Transactions on Database Systems (TODS)*, vol. 37, no. 2, p. 10, 2012.

[30] F. Chen and G. Roşu, "Parametric trace slicing and monitoring," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 246–261, Springer, 2009.

[31] S. V. W. Group *et al.*, "Systemc verification standard specification version 1.0 e," *SystemC Verification Working Group*, 2003.

[32] G. De Giacomo and M. Y. Vardi, "Linear temporal logic and linear dynamic logic on finite traces," 2013.

[33] M. dAmorim and G. Roşu, "Efficient monitoring of ω-languages," in *Computer Aided Verification*, pp. 364–378, Springer, 2005.

[34] A. Duret-Lutz and D. Poitrenaud, "Spot: an extensible model checking library using transition-based generalized büchi automata," in *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004.(MASCOTS 2004). Proceedings. The IEEE Computer Society's 12th Annual International Symposium on*, pp. 76–83, IEEE, 2004.

[35] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, *Aspect-oriented programming*. Springer, 1997.

[36] O. Spinczyk, A. Gal, and W. Schröder-Preikschat, "Aspectc++: an aspect-oriented extension to the c++ programming language," in *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, pp. 53–60, Australian Computer Society, Inc., 2002.

[37] D. Tabakov, G. Kamhi, M. Y. Vardi, and E. Singerman, "A temporal language for systemc," in *Formal Methods in Computer-Aided Design, 2008. FMCAD'08*, pp. 1–9, IEEE, 2008.