Assertion-Based Flow Monitoring of SystemC Models

Sonali Dutta
Rice University
Email:Sonali.Dutta@rice.edu

Moshe Y. Vardi
Rice University
Email:vardi@cs.rice.edu

***Abstract*—SystemC is the de facto standard system-modeling language for hardware-software systems. 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 process to another. Flows are dynamic and concurrent; a flow can begin anytime during the simulation and the system can process multiple flows at the same time.**

We provide a library for explicitly implementing flows in a SystemC model or annotating flows in an existing SystemC model with minimal modification. We also provide an automated monitoring framework for monitoring properties of flows. Such properties capture the reactive nature of a system naturally and are intuitive to write. Our experimental results show that the framework adds minimal simulation runtime overhead.

I. INTRODUCTION

SystemC (IEEE standard 1666-2005) has emerged as the de facto standard for modeling hardware-software systems [11]. SystemC is implemented as a C++ library, which defines macros and base classes for modeling concurrent hardware elements at various abstraction levels [14]. As a high-level, unifying modeling language, SystemC enables co-design and co-simulation of hardware and software, early in the system design cycle.

A typical SystemC model models the components of a system using SystemC modules. Each module can have one or more SystemC processes, with concurrent semantics. The SystemC library is accompanied by SystemC simulation (OSCI) kernel, see http://www.accellera.org/downloads/standards/systemc. In a simulation, the processes are scheduled in an interleaving fashion by the kernel, and executed on a single processor. Execution of a SystemC model consists of three phases in order: *elaboration phase*, *simulation phase*, and *cleanup phase* [1]. 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 the SystemC processes is simulated by the kernel. The cleanup phase can be used by the user to analyze the output of the simulation phase.

The growing popularity of SystemC has motivated research in functional validation of SystemC models [9], [18]. A particular direction is that of *assertion-based dynamic verification* (ABDV) of SystemC models [21]. ABDV involves three steps: (1) describing the behavior of the model under verification (MUV) in terms of assertions, (2) generating run-time monitors from input assertions, and (3) executing the MUV with those monitors, which observe the execution and report if the observed execution satisfies the specified assertions.

For example, in the ABDV framework developed by Tabakov and Vardi [19]–[22], an assertion is a temporal property of the simulation trace. We call it a *trace property*. Before executing the MUV, one C++ monitor class is generated from each assertion. During the monitored simulation of the MUV, one instance of each monitor class is created in the elaboration phase. In the simulation phase, those monitor instances are executed with the MUV. The number of monitor instances is equal to the number of assertions. There are three possible outcomes of a monitor: PASS, FAIL, and UNDETERMINED. In principle, trace properties are interpreted over infinite traces. But we can only monitor a finite prefix of an infinite trace. If the assertion contains any future obligation that is not met during the finite simulation, the monitor outputs UNDETERMINED. CHIMP [7] is a tool that implements this framework.

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* [10]. 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. In a concurrent and reactive hardware-software system, a flow can be a job submitted from outside the system or a job generated by some internal component. Each flow begins with a set of

Work supported in part by NSF grants CNS 1049862 and CCF-1139011, NSF Expeditions in Computing project "ExCAPE: Expeditions in Computer Augmented Program Engineering", BSF grant 9800096, and a gift from Intel.

input data, flows through different components in the system, and then ends producing output data. Thus, a flow represents both a flow of control and a flow of 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. Section II shows an example illustrating how flows crosscut different components of a system. Each flow type is associated with a set of data variables containing the inputs to the flow, the outputs produced by the flow, and any intermediate data. We call them *flow attributes*. For example, flow attributes associated with 'money withdrawal' flow type are card number, pin, amount to withdraw (input attributes), transaction status (output attribute), initial balance, and final balance (intermediate 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. Since the system is concurrent, at any point during the simulation there can be multiple live flows of same or different types.

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. Our goal here is to enable assertion-based monitoring of flow properties, as these capture well the behavior of reactive systems. 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.

The challenges are the following. To monitor a flow individually, the monitor has to know when a flow starts and ends, what its attributes are, and how the attributes can be accessed. This important information has to be explicit and accessible to the monitors during 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 attributes, beginning and ending a flow, and passing a flow from one SystemC process to another. Also, 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 extends trace-property monitoring.

The contribution of this paper is two fold. First, 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, 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. Section IV describes this library in detail. Second, we describe a framework for assertion-based dynamic verification of flow properties in a flow model. This includes (1) an algorithm to generate a flow monitor class from a flow property and a tool called FLOWMONGEN that implements it, and (2) a decentralized and dynamic algorithm to monitor dynamic and concurrent flows, and (3) an automated and efficient software framework to monitor flows in a flow model. The algorithms are described in Section V and the framework is presented in Section VI.[1] A case study and experimental results, presented in Section VII, show that our framework puts minimal runtime overhead of monitoring.

II. FLOWS: A DETAILED EXAMPLE

We present here a detailed example showing how flows of a flow type flow through the components of a system. Fig. 1[2] shows the different possible paths that a 'money withdrawal' flow 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. 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 goes concurrently 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 the 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. 1, if the input pin is incorrect, the flow takes the red path, otherwise it takes the green path. Irrespective of what path the flow takes, it must satisfy all 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 ACCEPT; else eventually globally it will be REJECT"[3]. Note that the current balance here is a global variable, stored in the bank database.

[1]Software tool is available at http://systemcflow.sourceforge.net

[2]The figure is best viewed online for color differentiation.

[3]We use Linear Temporal Logic to specify flow properties; see Section V.

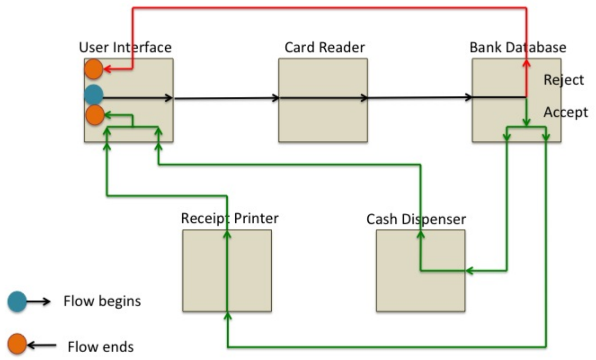

Fig. 1. Flows of type 'money withdrawal' crosscut different components of ATM server.

III. Related Work

The discussion of related work can be divided into two parts: work related to modeling flow-like entities and work related to monitoring of flow properties. There are several existing concepts similar to flows, such as transactions in μ-TLM (Transaction-Level Micro-architecture model) [15], [16], message flows [23], LSCs (Live Sequence Chart) [3], workflows [13], and behavioral programs [10]. All of these are modeling frameworks that 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 [12] 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 [17]. 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.

Prior work that is closest to our focus on monitoring flows are parametric monitoring of Java in MOP (Monitor-Oriented Programming) [2] and transaction monitoring in SCV (SystemC Verification library) [8]. MOP does decentralized and dynamic monitoring for parametric properties for Java. An example of a parametric property is: a vector cannot be modified when one of its enumerations is being used. This property is automatically verified for all vectors defined in the Java program under verification. Our dynamic and decentralized flow monitoring algorithm is influenced by MOP. SCV provides several constructs and APIs for verification of SystemC models. SCV allows users to define transaction types and capture transaction-level activities during simulation for monitoring. Yet MOP and SCV do not have our notion of "monitor-able" crosscutting flows that can branch and merge.

IV. Flow Library

We provide a light-weight (228 LOC) C++ library, called Flow Library, using which one can design flow models from scratch or annotate flows in an existing SystemC model. In this library, a flow type is a class and its attributes are the class member variables. A flow is then simply an instance of a flow-type class; to begin or end a flow, we just create or destroy the flow object.

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 (flow_id). So Flow Library provides `flow` class as the base class of all user-defined flow-type classes. The type_id of a flow type is passed to the flow class constructor by the constructor of the flow type class. The `flow` class contains three non-static member variables: `flow_id`, `type_id` (all the flows of same type has same type_id), and `num_proc`. The `flow` class automatically assigns a unique `flow_id` to each flow when it begins. Since a flow can branch, it can be concurrently processed by several SystemC processes. To avoid accidental ending of a flow by a process while some other process is still using it, it is important to keep track of the number of processes processing a flow at any point in time; `num_proc` automatically keeps track of that.

The flow model of ATM server in Fig. 1 has three flow types: 'money withdrawal', 'check deposit', and 'balance inquiry'. Assume that the type_ids of these flow types are 0, 1, and 2 respectively. The listing below shows how to model the class for 'money withdrawal' flow type. In addition to defining the flow attributes, this class provides setter and getter functions for accessing these attributes. (As shown later, these setter functions can be used to automatically call the monitors when a flow attribute changes its value.)

```
#include "flow.h"
...
enum trans_status{NOT_ASSIGNED, ACCEPT, REJECT};
class money_withdrawal : public flow{
      public:    //Constructor
        money_withdrawal(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*/
        ...
    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;
}; //End of class
```

To monitor flows, the monitors need to know when a new flow begins and ends, how to access its attributes, etc. The cleanest way to expose this information to 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 flows without interacting with every flow directly. For that Flow Library provides a class `flow_manager` that keeps all information about flows and provide them to monitors. The flow_manager class maintains a data structure, called *alive_flows*, which contains the list of all live flows.

A flow flows from process to process. Each process does some operation on the attributes of the flow and updates the global state (cf., [19]) of the system accordingly. Such processes usually run forever synchronizing with some user-defined clock. At every clock cycle, a process may operate on some flow object. Operating on a flow consists of the following three steps in sequence: (1) begin a new flow or get a flow from another process, (2) read and write the attributes of that flow and update the state of the system accordingly, (3) either end the flow or pass the flow to another process.

The `flow_manager` class provides four APIs that can be used by the SystemC processes to register new flows with flow_manager and access those flows from anywhere in the model. They are: (1) `begin_flow`: to begin a new flow; (2) `end_flow`: to end a live flow; (3) `get_flow`: to access a flow object, given its `flow_id`; (4) *release_flow*: to release a flow. (A process informs the flow_manager that it is done operating on a flow by releasing that flow.)

Each flow model defines a global pointer variable *flow_manager* fmanager* that points to an object of flow_manager class. *fmanager* is used by the processes to invoke the above APIs. Flow Library also does some automatic error checking and each API returns an error code. There are six error codes defined as *enum error_code* in Flow Library (See Table I).

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 I. ERROR CODES DEFINED IN ENUM ERROR_CODE

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` checks if f is indeed a new flow; if not, ALREADY_ALIVE is returned; if f is a null pointer, NULL_FLOW is returned. Else, `begin_flow` inserts f in `alive_flows` and returns DONE. The following listing shows how to model the process in the ATM User-Interface module that begins a 'money_withdrawal' flow at every clock cycle and sends it to a process of card-reader module.

```
void user_interface::submit_requests(){
  while(1){
    /*Decide which type of request
    to submit. Options are money withdrawal,
    check deposit and balance inquiry.*/
    ...
    if(request_type == "money_withdrawal"){
     /*choose the amount to withdraw*/
      unsigned int a = ... ;

      //Begin a "money withdrawal" flow
      error_code e;
      money_withdrawal* f=new money_withdrawal(a)
      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
```

To end flow f, a process calls *end_flow* with id of f: fid.

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

If fid is not the flow_id of an alive flow, NOT_ALIVE is returned; if some other process is still using the flow, CANNOT_END is returned. 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 the User-Interface module that ends flows of type 'money_withdrawal' (flows that did not get rejected at bank database). Note that ending also involves merging of the flow from receipt printer and cash dispenser.

```
void user_interface::end_money_withdrawal_success
  while(1){
    /*Get the id of the next flow
    sent by receipt_printer*/
    unsigned int id = ... ;
    /*Get the corresponding flow pointer*/
    money_withdrawal* 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
```

A process can also transfer, branch and join a flow. To maximize flexibility, Flow Library does not provide direct APIs for these operations but leave them as user defined. We show below how a process in a flow model can transfer a flow to another process, branch a flow (sending it to multiple processes), or merge a flow from multiple processes using Flow Library.

After a process finishes operating on a flow, it can transfer the flow to another process. The transfer happens through shared elements like global FIFO queue, TLM FIFO channels, and the like. To avoid unintended mishandling of pointers, it is strongly recommended to transfer only the flow_id rather then the 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 as follows:

```
//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);
```

If f is not alive, *get_flow* assigns NOT_ALIVE to e and returns 0, else it increments f->num_proc by one, assigns DONE to e and returns pointer to f. After a process finishes reading and writing to the attributes of f, it must release the flow using the *release_flow* API as:

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

If f has already ended, *release_flow* returns NOT_ALIVE, else it decrements f->num_proc by one and returns DONE. If num_proc is already 0, it returns CANNOT_RELEASE instead. After releasing a flow, a process must not use the flow pointer. 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;
    money_withdrawal* f = (money_withdrawal*)
        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
```

When two or more steps of the flow are independent of each other, two or more processes can concurrently work on the same flow. This is called branching and those processes are called 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 can operate on the flow concurrently. In Fig. 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 = ... ;
    money_withdrawal* f = ... ;

    if(!((f->get_accoun_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
```

A join can happen only after all the branch processes finish operating on the branched flow. The process where the flow joins must wait until then. This synchronization can be easily implemented using SystemC events and flow attributes. See the end_flow listing above to see how a 'money withdrawal' flow joins in the user module. Thus, a join is implicit and must happen after the branch and before the flow ends.

V. Flow Algorithms

A. Flow-Monitor-Generation Algorithm

The trace π_f of a flow f is a finite slice of the simulation trace; π_f begins and ends when f begins and ends. Traces of concurrent live flows overlap. A flow property is interpreted over the finite trace of a flow. The language we in which express flow properties is LTL_f*–Linear Temporal Logic* (LTL) interpreted over finite traces [4]–whose semantics is the obvious adaptation of LTL to finite traces. A flow f of type T satisfies a flow property P associated with T if the trace π_f of f satisfies P. Recall that trace properties are interpreted over infinite traces; since a simulation is always finite, monitoring a trace property may yield PASS, FAIL, or undetermined. In contrast, flow properties either hold or do not hold in a finite traces. Thus, monitoring flow properties yield only PASS or FAIL for all completed flows. (When simulation ends we also report all incomplete flows.)

In this work we focus on *intraflow* properties, where each property refers only to the attributes of a single flow. Generally, intraflow properties cannot express interaction between flows, though they can capture the interaction between the flow and the system components by referring to global variables. We convert each flow property to a C++ monitor class. Following [20], each monitor is a C++ encoding of a DFA (Deterministic Finite Automaton). The transition function of the DFA is encoded as a *step()* function in the monitor class. Calling that step() function once means making one transition in the DFA. We do not use accepting states in the monitors. A monitor rejects by finding no possible transition from some state. If it does not reject by the time the flow has ended, it accepts.

Tabakov et al. [20] describe how to generate C++ monitor class from DFAs; we use their algorithm here (there are several possible encodings of DFAs as C++ monitors, we use the *front_det_ifelse* encoding). To generate DFAs from LTL_f formulas we use the SPOT tool [5]. SPOT provides an optimized implementation of LTL to DFA conversion, but it uses LTL, which is over infinite traces. To get DFAs for LTL_f, we use a reduction from LTL_f to LTL. Our flow-monitor generation algorithm is the following:
Input: flow property φ (LTL_f formula)
Output: C++ flow monitor class M_φ
Steps:

1) From φ, generate the LTL formula φ' as $\varphi' = g(\varphi)$ (function g is defined below).
2) Use SPOT to Convert φ' to DFA A_φ that rejects minimal bad prefixes of φ' [20].
3) Encode A_φ in C++ monitor class M_φ [20].

The transformation function $g : LTL_f \rightarrow LTL$ was proposed by De Giacomo and Vardi in [4] as a reduction of LTL_f satisfiability to LTL satisfiability. It is defined as: $g(\varphi_f) = t(\varphi_f) \wedge (aliveUG!alive)$. The function $t : LTL_f \rightarrow LTL$ is inductively defined as:

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

The above transformation adds a new atomic proposition $alive$, which does not occur in φ_f. Intuitively, $alive$ is true until the finite trace ends and then it becomes false forever. This expected behavior of $alive$ is expressed by the formula $(aliveUG!alive)$.

The tool FlowMonGen implements the flow-monitor-generation algorithm. The input is a configuration file where the user can define a set of flow properties for each flow type, defined in the MUV. The output will be one flow-monitor class per flow property. FlowMonGen also generates a class called local_flow_manager, which instantiates the flow-monitor classes and executes those instances according to the flow-monitoring algorithm discussed below.

B. Flow-Monitoring Algorithm

Our goal is to verify that each flow f of type T satisfies each flow property P, associated with T. The flow-monitoring algorithm is as follows. For each flow f of type T and flow-monitor class M_P associated with property P of flow type T, do the following:

1) When f begins during the simulation, a monitor instance m_f^P of M_P is created and assigned to f.
2) While f is alive, execute the step() function of the monitor m_f^P with $alive = \mathbf{true}$.
3) If m_f^P rejects before f ends, record monitoring status FAIL and delete m_f^P.
4) Else:
   a) When f ends, execute once the step() function of m_f^P with $alive = \mathbf{false}$.
   b) If m_f^P rejects, record monitoring status FAIL, else record monitoring status PASS.
   c) Delete m_f^P.

For proof of the following theorem, see [6].
Theorem: m_f^P ends with PASS iff the flow f satisfies the flow property P.

The flow-monitoring algorithm is dynamic, because all flow-monitor instances are created during simulation, synchronized with the start of new flows. It is decentralized, because there is one monitor instance *per flow* and *flow property* pair, in contrast to one monitor instance per *trace property*.

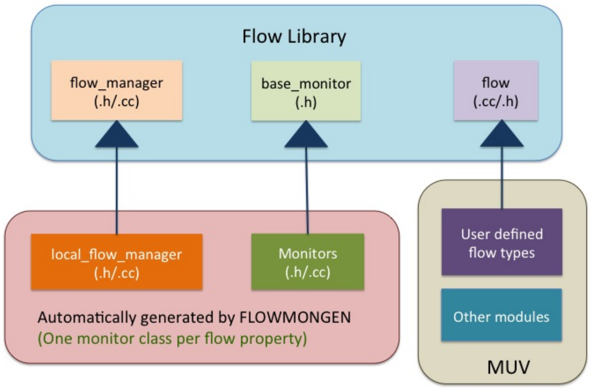

Fig. 2. Three components of flow monitoring framework

VI. Flow-Monitoring Framework

The flow-monitoring framework implements the dynamic and decentralized flow-monitoring algorithm. Fig. 2 shows the three components of this framework: (1) Flow Library, (2) SystemC model under verification (MUV), and (3) flow-monitor classes (one per flow property), along with the local_flow_manager class, generated by FlowMonGen.

The flow-monitor classes are derived from the base_monitor class of Flow Library. The transition function of a monitor DFA is captured in the step() function of a flow monitor-class (see [20]). Every time step() is called, a monitor instance makes a transition from one state to another depending on the current state of the system and the flow it is monitoring. If no such transition is available, step() of that monitor instance rejects.

The local_flow_manager class is derived from the flow_manager_class. When a user process begins a flow, it informs the flow_manager class by calling its API. Then flow_manager class calls an API of local_flow_manager class to inform it about the newly begun flow. Ending a flow is handled similarly. The alive_flow data structure of the flow_manager class is defined as a protected member variable so that the derived class local_flow_manager can access the attributes of all alive flows. The complete information about which flow-monitor class is associated with which flow-type is hardcoded in local_flow_manager class. During the monitored simulation, local_flow_manager dynamically creates the flow-monitor instances, executes their step functions, and deletes them, synchronizing with start and end of different flows.

The user can run the MUV with monitors (monitored simulation) or without monitors (unmonitored simulation). We want to minimize the modification of MUV to run in monitored and unmonitored mode. Also, the unmonitored simulation should not put any monitoring overhead. For that, the local_flow_manager class overrides some virtual methods of flow_manager class to do some extra work during monitored simulation. More precisely, in monitored mode, the global pointer *flow_manager* fmanager* points to a local_flow_manager object, whereas in unmonitored mode, it points to a flow_manager object, as shown below:

```
//For unmonitored simulation
flow_manager* fmanager = 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 );
```

The local_flow_manager performs the steps of the flow-monitoring algorithm. When a new flow f of type T begins, local_flow_manager automatically creates one instance of each flow-monitor class associated with T. These instances are responsible to monitor f. step() of a monitor instance of f is executed one or more times until step() rejects or f ends. If step() rejects, local_flow_manager deletes the corresponding monitor instance after recording its status as FAIL. When f ends, local_flow_manager executes step() of f's live monitors with $alive$ = false. If a monitor's step() rejects at this point, then local_flow_manager records that monitor's status as FAIL, else as PASS. Then it deletes all the monitor instances of f.

The flow-monitoring framework provides the following two APIs to execute step() of a monitor instance.

```
//Executes step() of monitor instances of f
//fid : ID of flow f
fmanager->monitor_flow(fid);

/*Executes step() of all monitor instances
of all live flows.*/
fmanager->monitor_all();
```

Both these APIs do nothing in unmonitored mode. Calling the second API is more expensive, since it executes the monitors of *all* live flows.

It remains to discuss when to execute step() of a monitor instance. In the terminology of [19], the question is when to sample the trace. A naive approach would be to execute step() of all live monitor instances following execution of each statement of the MUV. But this can be very expensive, because in flow monitoring the number of monitor instances are not equal to the number of flow properties, but is equal to the number of flow properties multiplied by the number of live flows at that point. This may lead to the execution of thousands of monitor instances after every MUV statement, which would incur significant runtime overhead. This method is not only inefficient, but also redundant. A flow changes its state only when a new value is written to one of its flow attributes. Not all of those thousands flows change their state after execution of each MUV statement. So running the monitors of a flow when nothing about it has changed is redundant. One can use customized sampling, as in [21], but that requires nontrivial user effort.

A better approach is to track when a flow attribute is changing its value, and, when it does, execute only the monitor instances of that flow. This ensures that the step() functions are executed frequently enough to capture all state changes of the flows (completeness). Also, we do not monitor a flow when it does not change its state (no redundancy).

For that, Flow Library provides a function *set_attribute* in flow class. Suppose that the MUV has a flow type T that has a flow attribute *int a*. Let f be a flow of type T. To automatically execute all flow-monitor instances of flow f, every time the flow attribute a of flow f is written, the setter function of a in flow-type class T has to be defined as follows:

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

set_attribute() is defined in the flow class 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 our example, int). In unmonitored mode, this function just sets the value of the attribute. In monitored mode, after setting the value, the setter also executes one step of all flow-monitor instances of the flow, whose attribute's value has been set. Similarly, to monitor all flows when an important global variable changes its value, call *monitor_all()* inside the setter function of that global variable. Both *monitor_flow()* and *monitor_all()* can be called directly from any place in the MUV.

Since kernel phases are important temporal locations in SystemC simulation, the framework also supports automatic execution of step() at 18 different kernel phases (defined in [19]). In the input file to FLOWMONGEN, the user can define a flow property to be *sensitive* to a set of kernel phases. During the monitored simulation, when a kernel phase occurs, all monitor instances sensitive to that phase are executed by the local_flow_manager automatically. For example, suppose that flow monitor class M, associated with flow type T, is sensitive to kernel phase t_1 and t_2. Then all monitor instances of M will execute whenever any of t_1 or t_2 occurs during monitored simulation. It is important to note here is that at every occurrence of t_1 and t_2, all monitors instances of M, assigned to all currently live flows of type T, are executed (irrespective of if the flow has changed its state or not). Thus, monitoring at kernel phases is more expensive than monitoring at value change of attributes. The experimental results, described in Section VII, support this too.

To execute the flow monitors at different kernel phases, the local_flow_manager has to know when a kernel phase occurs. In the standard implementation of the OSCI kernel, this information is not exposed. So, following the patch described in [21], we have put a minimal patch on SystemC kernel to expose those information. Our SystemC patch is only 158 LOC as compared to the patch in [21], which is 1100 LOC. Also our patch supports both static (trace property) and dynamic (flow property) monitoring. (The patch in [21] does not support dynamic monitoring.) Our patch is easily portable to future SystemC releases.

VII. EXPERIMENTAL EVALUATION

To assess the flow-monitoring framework, we performed a case study, where we designed a flow model and verified some flow properties of it using the framework. We took the Airline Reservation System (ARS) model, developed by Tabakov and Vardi [21], and modified it to make the idea of flows explicit.[4] The model was originally 3,100 LOC; turning it into a flow model required insertion of an additional 70 LOC. This shows that converting an existing SystemC model to a flow model takes minimal effort and by doing so, one can monitor flow properties, includes liveness properties that are not monitorable in trace property monitoring.

ARS models a multi-user, interactive, concurrent system for purchasing airline tickets. It has one flow type: `request` to reserve a trip, which can be 1-or 2-way trip. The input attributes are `source`, `destination`, `date1` (the date of flying from source to destination), `date2` (date of flying from destination to source for return-trip requests), `is_return` (true for return-trip requests), `seats` (number of people traveling), and `category` (economy or business class). The output attributes are `trip1` (sequence of connecting flights from source to destination) and `trip2` (sequence of connecting flights from destination to source for return-trip requests). There is one intermediate attribute, called `speculative_bit`, whose functionality is explained later.

ARS contains four SystemC modules: a user module that simulates the users, and three system modules (IO, master and planner) that process the requests from the user. There are two different clocks defined in ARS: `user_clock` and `system_clock`. The user module operates by `user_clock` and submits one new request at every clock cycle. The system modules operate by `system_clock`. At every `system_clock` cycle, a system process does the following: it fetches a new request from some shared element, process the request, and send it to some other shared element. In ARS, the shared elements are either bounded queues or channels. Two processes from the same module communicate through a bounded queue, defined internally in that module. Two processes from two different modules communicate through a bounded channel connecting those modules. The sum of the capacities of these shared elements puts a bound on how many requests can be concurrently alive in the system at any point of time during the simulation. The processes use SystemC events to synchronize reading and writing to the shared elements.

Fig. 3 shows how a request flows through ARS model. A request r is generated by a process in the user module. Then it travels to the IO module. If the bounded queue in IO module has no space to store the new request, the request returns to the user module and ends. (This happens when `user_clock` is faster that `system_clock`, as more requests are getting generated than the system can process.) Otherwise, IO module sends the request to the master module. If it is a 1-way trip request, the master module sends it to the planner module. The planner module finds the connecting flights with a probability of 90%, and then sends the request back to master, who sends it to IO, and, finally, IO sends it to the user, where the request r ends. If r a return-trip request, then master branches r to process the two trips separately. (This is a way to test that our framework supports branching and merging of flows.) The `speculative_bit` of r is used to synchronize the branching and merging. The `speculative_bit` is set true when r begins. First `trip1` of r is found and then `trip2`

[4]The original and modified models are available at http://www.cs.rice.edu/~vardi/ARS_models.zip.

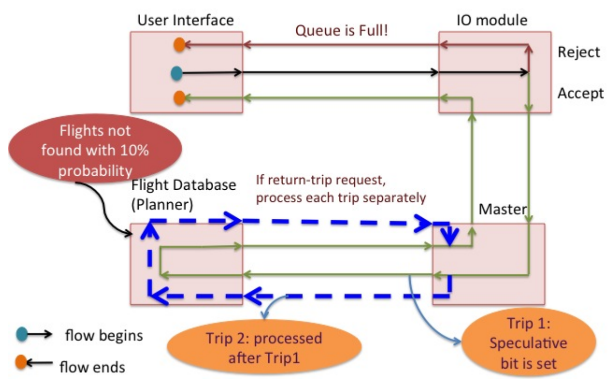
User Interface
Queue is Full!
IO module
Reject
Accept
Flights not found with 10% probability
Flight Database (Planner)
If return-trip request, process each trip separately
Master
flow begins
flow ends
Trip 2: processed after Trip1
Trip 1: Speculative bit is set

Fig. 3. Possible paths taken by a "request" in ARS model

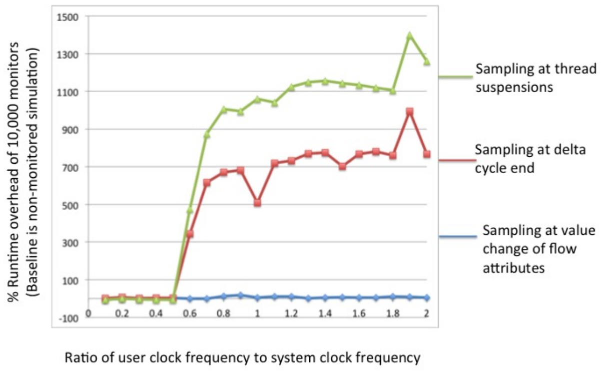

Fig. 4. Variation of flow monitoring overhead of Property 1 with respect to ratio of user to system clock frequencies

is found. Once `trip2` is found, `speculative_bit` is set false. After planner finds both the trips, the request goes back to the user, following the same path it came through.

We have verified three flow properties of ARS. Each property is verified for all the requests that are submitted by the user module during the simulation. Property 1 is a liveness property that says that eventually trip1 should be found and if it is a return trip request, then eventually trip2 should be found. The requests that are sent back to the user by the IO module due to unavailability of space in internal queues do not satisfy this property. Also the requests for which planner does not find connecting flights do not satisfy this property. Property 2 is another liveness property that captures the behavior of `speculative_bit` of all return trip requests. Property 2 says that if it is a return-trip request, then eventually `speculative_bit` should be true, and then eventually globally it should become false. This property is not satisfied by the return-trip requests that are sent back to the user due to unavailability of space, but every other return-trip request should satisfy this property. Property 3 is a safety property that says that the number of connecting flights of any trip should not exceed the maximum number of legs defined in the planner module. Notice that Property 3 refers to a global variable, which is in the planner module.

A major concern in monitoring is the overhead that the monitors put on the execution of the MUV. The monitoring overhead is defined as the increase in the runtime in monitored simulation, compared to its unmonitored version. The runtime overhead of flow monitoring depends on the number of active monitors and how often they are activated. Thus, the overhead may depend on multiple factors: the relative frequency of user clock and system clock, the system capacity (the maximum number of concurrent live flows that the system can hold at any point during the simulation), and the sampling rate of the monitors (how frequently the function step() is executing). In our experiments, we wish to find out how runtime overhead is affected by the above mentioned factors.

When user clock is faster than system clock, requests start getting buffered in the shared elements. With the increase in the ratio of user-clock frequency to the system-clock frequency, more and more requests start filling up the shared elements until they overflow and requests start to fail (when IO sends requests back to user without processing). More pending requests means higher number of concurrent live flows in the system. Also, if we increase the system capacity by making the shared elements larger, less requests would fail but the number of pending requests would increase. So increase of either clock ratio or system capacity would result in the increase of the number of concurrent live requests in the system. To see how the number of concurrent live requests affect the monitoring overhead, we have to consider how frequently we are executing the flow monitors (sampling rate). As stated before, we have two types of sampling: sampling at value change of flow attributes and sampling at kernel phases. Let us consider monitoring overhead for both sampling modes.

When sampling at value change of flow attributes, the framework only executes the monitors of the flow whose attribute value has changed. Suppose there are n flows that execute during the simulation, each with k attribute assignments and one monitor instance associated with it. Then the total number of step() function calls are kn. This number does not depend on the number of concurrent live flows in the system. So, monitoring overhead when sampling at value change of attributes does not change with system capacity or clock frequency ratio.

In contrast, when sampling at kernel phases, at every kernel phase the framework executes all monitors instances of all live flows. So, the total number of step() executions increases with the number of concurrent live flows in the system, which increases the monitoring overhead. As discussed above, the number of concurrent live flows increases with the increase of frequency ratio or the system capacity (which is proportional to the capacity of each queue in ARS). So while sampling at a kernel phase, the runtime overhead increases with the increase of clock frequency ratio and the system capacity.

The above discussion is validated by the experimental results, shown in Fig. 4 and Fig. 5. For each simulation, total simulation time is 100,000 SC_NS and a new flow is generated after every 10 SC_NS, yielding 10,000 flows. The results shown are for Property 1; Property 2 and Property 3 yield similar results. Each data point in the plots is the average of 100 simulations. In both graphs, we observed the runtime overhead at three sampling rates by sampling at: value change of flow attributes, kernel phase *delta cycle end*, and kernel phase *thread prices suspend*. The baseline is the runtime of unmonitored simulation with the same value of system

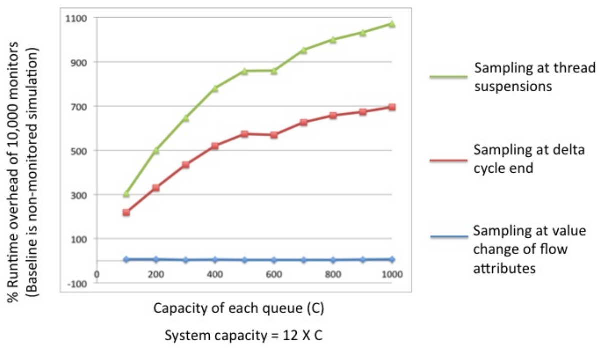

Fig. 5. Variation of flow monitoring overhead of Property 1 with respect to the system capacity (maximum number of simultaneous live flows the system can store)

capacity and clock frequency ratio.

In Fig. 4 we varied the frequency ratio (user / system) from 0.1 to 2 by varying the frequency of system clock and observed the runtime overhead at the three sampling rates. As expected, the runtime overhead does not change with the clock frequency ratio while sampling at value change of attributes, but does increase while sampling at kernel phases. In Fig. 5 we varied the system capacity from 100*12 to 1000*12 and observed the runtime overhead at the three sampling rates. As expected, the runtime overhead does not change with the system capacity while sampling at attributed-value change, but does increase with the system capacity while sampling at kernel phases. The graphs show the overhead for 10,000 monitors. So the overhead per monitor is minimal.

VIII. Conclusion

We have introduced here the concept of flows and flow-based monitoring in SystemC. Flow properties are the suitable candidates to capture the reactive and job-oriented behavior of many hardware-software systems. Using Flow Library one can make any SystemC model a flow model with minimal modification. Flow Library is lightweight and easy to use. The FlowMonGen tool automatically generates flow-monitor classes for user-provided flow properties. Using the dynamic and decentralized flow framework, one can automatically verify that every flow of a certain flow type satisfies each flow property associated with that flow type in a monitored simulation. The experimental results shows that the framework adds minimal runtime overhead.

In future work we plan to define hierarchies of flows. A flow can contain subflows. For example, to process an image, first divide it into smaller sub-images, then process those sub-images, and then combine the result. Processing the larger image is a parent flow and processing each sub-image is a subflow. Also we plan to capture interactions among multiple flows. Our current approach is focused on intra-flow properties. Capturing interaction among flows, would require us to consider inter-flow properties, such as "parent flow returns SUCCESS iff all its sub-flows return SUCCESS." Finally, it is interesting to use data-flow analysis to detect flows in legacy SystemC models.

References

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

[2] F. Chen and G. Roşu. Parametric trace slicing and monitoring. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 246–261. Springer, 2009.

[3] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[4] G. De Giacomo and M. Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 854–860. AAAI Press, 2013.

[5] 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*, pages 76–83. IEEE, 2004.

[6] S. Dutta. Assertion-based flow monitoring of SystemC models. Master's thesis, Rice University, 2014.

[7] S. Dutta, D. Tabakov, and M. Y. Vardi. CHIMP: a tool for assertion-based dynamic verification of SystemC models. In *International Workshop on Design and Implementation of Formal Tools and Systems, 2013. DIFTS'13*. ACM, 2012.

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

[9] 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*, pages 177–180. IEEE, 2004.

[10] D. Harel, A. Marron, and G. Weiss. Behavioral programming. *Communications of the ACM*, 55(7):90–100, 2012.

[11] 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*, pages 171–178. IEEE, 2006.

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

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

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

[15] 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*, pages 123–132. IEEE, 2007.

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

[17] S. Maoz, D. Harel, and A. Kleinbort. A compiler for multimodal scenarios: Transforming LSCs into AspectJ. *ACM Trans. Softw. Eng. Methodol.*, 20(4:18):1–41, 2011.

[18] L. Pierre and L. Ferro. A tractable and fast method for monitoring SystemC TLM specifications. *Computers, IEEE Transactions on*, 57(10):1346–1356, 2008.

[19] 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*, pages 1–9. IEEE, 2008.

[20] D. Tabakov, K. Y. Rozier, and M. Y. Vardi. Optimized temporal monitors for SystemC. *Formal Methods in System Design*, 41(3):236–268, 2012.

[21] 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*, pages 123–132. IEEE, 2010.

[22] D. Tabakov and M. Y. Vardi. Automatic aspectization of SystemC. In *Proceedings of the 2012 workshop on Modularity in Systems Software*, pages 9–14. ACM, 2012.

[23] 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*, page 10. IEEE Press, 2008.