Intermediate Verification Language for SystemC

(IVL)

Vladimir Herdt, Hoang M. Le, Daniel Große, Rolf Drechsler

8 March 2018

http://www.systemc-verification.org/sissi

Language Reference Manual
Version 1.0

University of Bremen

Contents

1 Introduction

The SystemC Intermediate Verification Language (IVL) has been introduced in [7] with the purpose of simplifying the verification process of SystemC program, by separating it into two independent steps. The idea is that first a *front-end* converts a SystemC program into an IVL program, which is then verified by a separate *back-end*. The IVL has been designed to be compact and easily manageable but at the same time powerful enough to allow the translation of SystemC designs. A back-end should focus purely on the behavior of the considered SystemC program. This behavior is fully captured by the SystemC processes under the simulation semantics of the SystemC kernel. Therefore, a front-end should first perform the elaboration phase, i.e. determine the binding of ports and channels. Then it should extract and map the design behavior to the IVL. Separating the verification process of SystemC programs into two independent tasks makes both of them more manageable. The properties of IVL and the resulting advantages include:

- Compact, intuitive and readable language: IVL has been designed in such a way that both manual and automatic transformations from SystemC are possible.
- Independent development of front-end and back-end: IVL enables to focus on the problem that the user wants to address.
- Open language and support: IVL is open and a free parser is provided. Moreover, all freely available benchmarks used by existing formal verification approaches for SystemC have been transformed into an extensive IVL benchmark set. This accelerates research in particular with respect to new formal approaches.

In the following the structure and key components of the IVL are discussed in more detail. We start by reviewing essential background on SystemC in Section 2. Then provide a high-level overview of the IVL in Section 3. The following two sections provide more details on the IVL syntax and semantic (Section 4) and simulation semantics (Section 5). Finally, in Section 6 we translate the SystemC fifo example to IVL and comment on the process.

2 Background on SystemC

In the following only the essential aspects of SystemC are described. SystemC has been implemented as a C++ class library, which includes an event-driven simulation kernel. The structure of the system is described with ports and modules, whereas the behavior is described in processes which are triggered by events and communicate through channels. A process gains the *runnable* status when one or more events of its sensitivity list have been notified. The simulation kernel selects one of the runnable processes and gives this process the control. The execution of a process is non-preemptive, i.e. the kernel receives the control back if the process has finished its execution or suspends itself by calling *wait()*. SystemC provides three

types of processes with *SC_THREAD* being the most general type, i.e. the other two can be modeled by using SC_THREAD. For event-based synchronization, SystemC offers many variants of *wait()* and *notify()* such as *wait(time)*, *wait(event)*, *event.notify(delay)*, *event.notify()*, etc.

- *wait(event)* blocks the current process until the notification of the event.
- *notify(event)* performs an *immediate notification* of the event. Processes waiting on this event become immediately runnable in this *delta cycle*.
- *notify(event, delay)* performs a *timed notification* of the event. It is called a *delta notification* if the delay is zero. In this case the notification will be performed in the next *delta phase*, thus a process waiting for the event becomes runnable in the next *delta cycle*.
- *wait(delay)* blocks the current process for the specified amount of time units. This operation can be equivalently rewritten as the following block { `sc_event e; notify(e, delay); wait(e);` }, where e is a unique event.

Additionally, the *suspend(process)* and *resume(process)* functions can be used for synchronization. The former immediately marks a process as suspended. A suspended process is not runnable. The *resume* function unmarks the process again. It is a form of delta notification, thus its effects are postponed until the next *delta* phase of the simulation. Suspend and resume are complementary to event-based synchronization. Thus a process can be suspended and waiting for an event at the same time. In order to become runnable again, the process has to be resumed again and the event has to be notified.

2.1 SystemC Simulation Semantics

The execution of a SystemC program consists of two main steps: an *elaboration* phase is followed by a *simulation* phase. During *elaboration* modules are instantiated, ports and channels are bound and processes registered to the simulation kernel. Basically elaboration prepares the following simulation. It ends by a call to the *sc_start* function. An optional maximum simulation time can be specified. The simulation kernel of SystemC takes over and executes the registered processes. Basically simulation consists of five different phases:

1. *Initialization*: Processes are made runnable.
2. *Evaluation*: A runnable process is executed or resumes its execution. In case of immediate notification, a waiting process becomes runnable immediately. This step is repeated until no more processes are runnable.
3. *Update*: Updates of channels are performed. These updates have been requested during the evaluation phase.
4. *Delta notification*: If there are delta notifications, the waiting processes are made runnable, and then the simulation is continued with the Evaluation step.

5. *Timed notification*: If there are timed notifications, the simulation time is advanced to the earliest one, the waiting processes are made runnable, and the simulation is continued with the Evaluation step. Otherwise the simulation is stopped.

The interested reader is referred to [3, 5, 2] or the IEEE standard [6] for more details on SystemC.

3 Overview

The IVL is the stepping stone between a front-end and a back-end. Ideally, it should be compact and easily manageable but at the same time powerful enough to allow the translation of SystemC designs. Our view is that a back-end should focus purely on the behavior of the considered SystemC design. This behavior is fully captured by the SystemC processes under the simulation semantics of the SystemC kernel. Thus, a front-end should first perform the elaboration phase, i.e. determine the binding of ports and channels. Then it should extract and map the design behavior to the IVL. The IVL is described in the following Based on the simulation semantics of SystemC, we identify the three basic components of the SystemC kernel: *SC_THREAD* (the other two SystemC processes can also be modeled by using SC_THREAD), *sc_event* and *channel update*. These are adopted to be *kernel primitives* of the IVL: *thread*, *event* and *update*, respectively. Associated to them are the following primitive functions:

- *suspend* and *resume* to suspend and resume a thread, respectively;
- *wait* and *notify* to wait for and notify an event (the notification can be either immediate or delayed depending on the function arguments);
- *request_update* to request an update to be performed during the update phase.

These primitives form the backbone of the kernel and follow precisely the simulation semantics of SystemC (see [6]). Other SystemC constructs such as *sc_signal*, *sc_mutex*, static sensitivity, etc. can be modeled using this backbone.

The behavior of a *thread* or an *update* is defined by a function. Functions which are neither *threads* nor *updates* can also be declared. Every function possesses a body which is a list of statements. We allow only assignments, (conditional) goto statements and function calls. Every structural control statement (*if-then-else*, *while-do*, *switch-case*, etc.) can be mapped to (conditional) *goto* statements by a front-end. Therefore, the representation of a function body as a list of statements is general and at the same time much more manageable for a back-end, e.g. it has no longer to handle (potentialy nested) scopes and can keep track of the active statement in a function using a single index (i.e. instruction pointer). The well-known LLVM-IR as well as the CBMC verification backend take a similar approach.

As *data primitives* the IVL supports Boolean and integer data types of C++ together with all arithmetic and logic operators. Furthermore, arrays and pointers of primitive types are also supported. Additionally, bit-vectors of finite width can

```
1  SC_MODULE(Module) {
2    sc_core::sc_event e;
3    uint x, a, b;
4
5    SC_CTOR(Module)
6      : x(rand()), a(0), b(0) {
7      SC_THREAD(A);
8      SC_THREAD(B);
9      SC_THREAD(C);
10   }
11
12   void A() {
13     if (x % 2)
14       a = 1;
15     else
16       a = 0;
17   }
18
19   void B() {
20     e.wait();
21     b = x / 2;
22   }
23
24   void C() {
25     e.notify();
26   }
27 };
28
29 int sc_main() {
30   Module m("top");
31   sc_start();
32   assert(2 * m.b + m.a == m.x);
33   return 0;
34 }
```

Figure 1: A SystemC example

```
1  event e;
2  uint x = ?(uint);
3  uint a = 0;
4  uint b = 0;
5
6  thread A begin
7   if x % 2 goto elseif
8     a = 0
9     goto endif
10   elseif:
11    a = 1
12  endif:
13 end
14
15 thread B begin
16   wait (e);
17   b = x / 2;
18 end
19
20 thread C begin
21   notify (e);
22 end
23
24 main begin
25   start;
26   assert (2 * b + a == x);
27 end
```

Figure 2: The example in IVL

be declared. This enables the modeling of SystemC data types such as *sc_int* or *sc_uint* in the IVL.

For verification purpose, the IVL provides the *assert* and *assume* builtin functions. More expressive temporal properties can be translated to FSMs and embedded into an IVL description by a front-end. Symbolic values of primitive types are also supported.

The IVL does not support dynamic instantiation of SystemC constructs, i.e. as already mentioned we require a front-end to evaluate the elaboration phase to extract the SystemC architecture. Support for C++ classes, function pointers and exceptions is missing. A front-end is required to map these features to the IVL primitives, e.g. by unpacking classes (as also demonstrated in the following SystemC example).

Example 1. a) SystemC Example: Fig. 1 shows a simple SystemC example. The main purpose of the example is to demonstrate the main elements of the IVL. The design has one module and three SC_THREADs A, B and C. Thread A sets variable a to 0, if x is divisible by 2, and to 1 otherwise (Line 13-16). Variable x is initialized with a random integer value on Line 6 (i.e. it models an input). Thread B waits for the notification of event e and sets $b = x \ / \ 2$ subsequently (Line 20-21). Thread C performs an immediate notification of event e (Line 25). If thread B is not already waiting for it, the notification is lost. After the simulation the value of variable a

and b should be x % 2 and x / 2, respectively. Thus the assertion ($2 * b + a == x$) is expected to hold (Line 32). Nevertheless, there exist counterexamples, for example the scheduling sequence CAB leads to a violation of the assertion. The reason is that b has not been set correctly due to the lost notification.

b) IVL Example: Fig. 2 depicts the same example in IVL. As can be seen the SystemC module is *unpacked*, i.e. variables, functions, and threads of the module are now global declarations. The calls to *wait* and *notify* are directly mapped to statements of the same name. The *if-then-else* block of thread A is converted to a combination of conditional and unconditional *goto* statements (Line 7-12). Variable x is initialized with a symbolic integer value (Line 2) and can have any value in the range of *unsigned int*. The statement *start* on Line 25 starts the simulation.

In short, the IVL is kept minimal but expressive enough for the purpose of formal verification. It aims to cover all benchmarks used by existing formal verification approaches for SystemC. It would only take little effort to adapt these approaches to support this IVL as their input language. That would lead to the availability of a checker suite for SystemC once a capable front-end is fully developed. We also refer to an IVL description as a SystemC design since both define the same behavior. A grammar and a parser for the IVL are provided at our website `http://www.systemc-verification.org/sissi`.

4 Syntax and Semantics

In the following the structure of the IVL is described in a *top-down* approach. This section first roughly describes how a program is structured. Subsequently, the following subsections describe the individual statements (i.e. instructions), expressions, and the language type system in more detail.

A program consists of a list of global declarations. These include variables, events, functions, threads and updates. Both functions and threads/updates contain a list of instructions. Statements use expressions and each expression can contain subexpressions (which are themselves expressions). For the internal representation of a program, it makes sense to use a tree structure. Fig. 3 shows a high-level overview of the internal representation of an IVL program as an AST.

In terms of structure, threads, updates and functions are almost identical. However, they differ in terms of their simulation semantics. So threads and updates are completely managed by the simulator. Unlike functions, they can not be called manually from within the program. In addition, they do not have any return value or accept parameters (a consequence of the fact that they can not be called manually).

A special function is *main*. It is the entry point of the IVL program (program execution starts at *main*). In terms of simulation semantics, it differs from both threads and functions. It is initially called automatically and can not be called manually. It must not contain a blocking instruction. The function *main* must be defined in each program. The actual simulation phase can be started within *main* with the command *start*.

Variable declarations may be created (as the only declaration) both globally and locally (i.e. within functions). Fig. 4 shows an example of the syntax for the existing

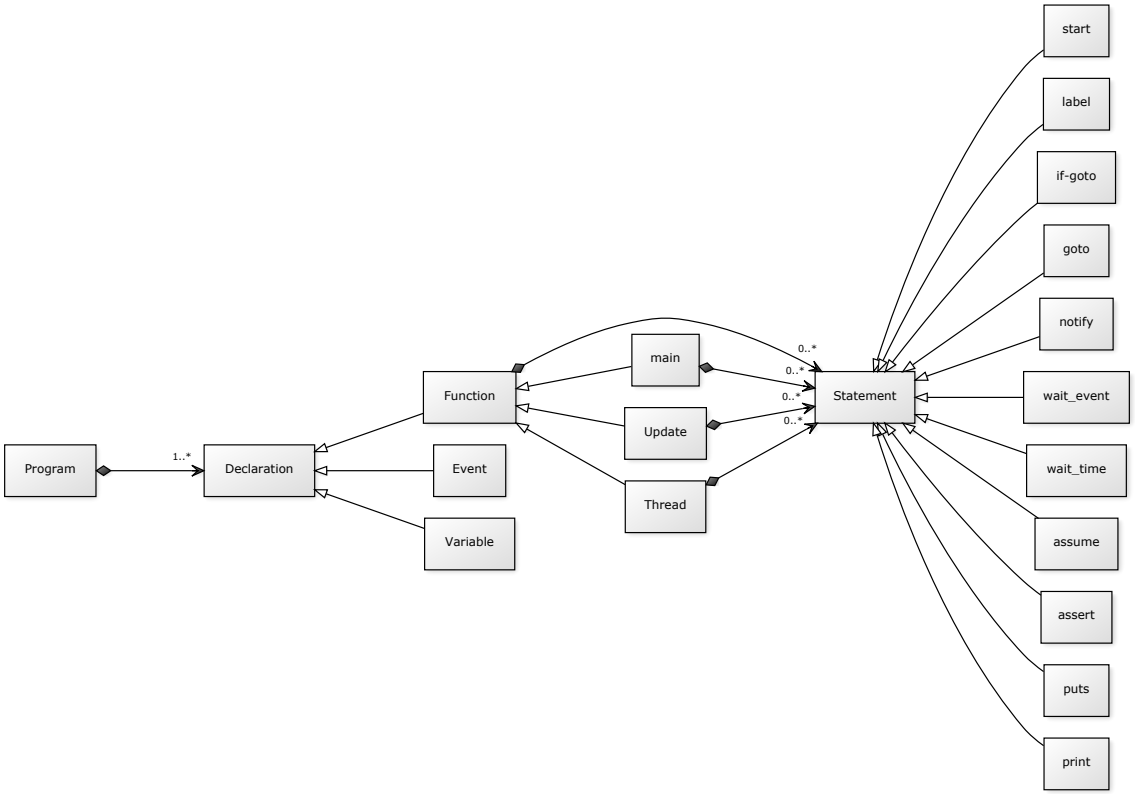


Figure 3: Top-level overview of the internal representation of the program as AST

declarations.

The individual statements of a function (or a thread) can be seperated either by a semicolon or a new line. In the case of a newline there should not be two statements in the same line. There can be any number of blank lines between individual statements and declarations.

A program is divided into several scopes (visibility areas for declarations). It has a global scpope and each function (or thread or update) opens a new local scope. Statements within functions may access the global scope, but not the other way around. All created declarations are added to the corresponding scope. Variables, events, functions, threads and updates are managed in separate namespaces, and it is always clear from the context which namespace to access if an identifier is to be resolved. Thus, e.g. declare a variable and a thread in the same scope with the same name, but it is not possible to declare two variables with the same name (in the same scope).

4.1 Reserved Identifiers

The following identifiers are used internally as keywords, built-in statements, and types, and should (or may be partially) not be used as identifiers for your own declarations (e.g. variable or function declarations):

```
1  /* Define a global variable with type int,
       name x and initial value 3 */
2  int x = 3
3  /* Events can only be global */
4  event e
5
6  /* A function declaration contains a name,
       a return type and a list of arguments
       */
7  int gen(int a, int b) begin
8    /* Variables can also be defined inside
         of function, the local variable
         shadows the global variable with
         the same name */
9    int x = a * 2
10   /* the result value has to be specified
         explizitly using a return statement
         */
11   return x + b
12 end
13
14 /* Defines a thread named check */
15 thread check begin
16   /* Similar to a function, a thread also
         contains a list of statements */
17   loop:
18     /* Blocks the thread execution until
           notification of event e is
           triggered */
19     wait e
20     assert gen(2, x) == 7
21   goto loop
22 end
23
24 thread clk begin
25   loop:
26     /* Blocks the thread execution for one
           time step */
27     wait_time 1
28     /* Trigger immediate notification of
           event e */
29     notify e
30   goto loop
31 end
32
33 main begin
34   /* Start simulation and run it for five
         time steps */
35   start 5
36 end
```

Figure 4: Example program to illustrate the syntax for different declarations

Table 1: Reserved identifiers

event	char	short	int	long	uchar	
ushort	uint	ulong	assert	puts	if	goto
begin	end	assume	thread	main	start	wait
wait_event	delay	wait_time	resume	suspend	signed	unsigned

4.2 Statements

Statements (i.e. instructions) are the building blocks of execution. Unlike expressions, they can not be nested. Instructions can be subdivided into the areas of control flow control, event handling, simulation control, information output and assertions. Fig. 5 shows the grammar of all available instructions.

The control flow can only be controlled as a combination of labels with conditional / unconditional jumps. The simple *goto* represents an unconditional jump to a label (unconditional jumps are always taken). In a conditional jump, the *goto* is preceded by *if* with a condition that must be fulfilled for the jump to be executed.

⟨*Statement*⟩ ::= ⟨*Expression*⟩ '`=`' ⟨*Expression*⟩
| '`assert`' ⟨*Expression*⟩ ⟨*String*⟩?
| '`assume`' ⟨*Expression*⟩
| '`suspend`' ⟨*Identifier*⟩
| '`resume`' ⟨*Identifier*⟩
| '`print`' ⟨*Expression*⟩
| '`puts`' ⟨*String*⟩
| ('`wait_event`' | '`wait`') ⟨*Identifier*⟩
| ('`wait_time`' | '`delay`') ⟨*Expression*⟩
| '`notify`' ⟨*Identifier*⟩ ⟨*Expression*⟩
| '`goto`' ⟨*Identifier*⟩
| ⟨*Identifier*⟩ '`:`'
| '`if`' ⟨*Expression*⟩ '`goto`' ⟨*Identifier*⟩
| '`start`' ⟨*Expression*⟩?
| '`return`' ⟨*Expression*⟩?

Figure 5: Instructions available in the IVL

The statement *return* represents a special unconditional jump. It terminates the currently executing function and returns to the caller. Optionally, a return value can be specified.

Goto statements allow to represent a more general control-flow compared to high-level control structures. Therefore, they are a reasonable choice for an intermediate representation that targets languages with goto support (such as C++). The well known LLVM-IR as well as the CBMC verification backend are also based on goto statements. Furthermore, this choice simplifies the back-end implementation as it has no longer to handle (potentialy nested) scopes and can keep track of the active statement in a function using a single index (i.e. instruction pointer).

The synchronization of threads is done via events. There are two basic operations defined on an event: *wait* and *notify*. *wait* blocks the currently running thread on an event. The blocking will be canceled as soon as the event is notified. A notification is performed by the *notify* statement. In addition to the event, it also receives an optional expression that specifies after how many simulation time steps the notification of the event should be triggered. If no time is specified then the notification will be immediate, with a time of zero a delta notification will be performed. The individual simulation phases will be described in more detail in Section 5. By means of the *wait_time* instruction, a thread is blocked for the given simulation time (and after the time has passed it becomes automatically runnable again).

The *suspend* statement suspends the given thread. A suspended thread is blocked and thus will not run. If *suspend* is applied to the currently running thread, its execution is interrupted and it is suspended. The *resume* statement removes the suspension of the given thread. The effect is always delayed, so only applied in the *delta-notify* phase. The instructions *suspend* and *resume* are independent of the event handling mechanism, i.e. a thread can be blocked on one event as well as suspended at the same time. Both effects have to be canceled for a thread to become runnable again.

The instructions *print* and *puts* are used to output information (for example, to understand the control flow). By means of *print* an expression is printed to stdout. The statement *puts* prints the given string. These two instructions do not modify the program state and thus can also be ignored in an implementation.

The actual simulation phase (see Section 5) is started with the *start* statement. Optionally, a maximum number of executed simulation time steps can be specified. Simulation will terminate once the maximum time is exceeded or no more thread is runnable.

The *assert* instruction introduces an assertion. At runtime, a check has to ensure that the given expression is valid. The possible values of a symbolic expression can be restricted by using the *assume* instruction. The (boolean) condition passed to *assume* is assumed to be valid for the entire further simulation process and thus constrains all subsequent evaluations.

4.3 Expressions

Expressions are defined recursively. The foundation for the recursive definition are the atomic expressions. Applying an operator to an existing expression returns again

⟨*Expression*⟩ ::= ⟨*Atom*⟩
| ⟨*Expression*⟩ '`||`' ⟨*Expression*⟩
| ⟨*Expression*⟩ '`&&`' ⟨*Expression*⟩
| ⟨*Expression*⟩ '`==`' ⟨*Expression*⟩
| ⟨*Expression*⟩ '`!=`' ⟨*Expression*⟩
| ⟨*Expression*⟩ '`<=`' ⟨*Expression*⟩
| ⟨*Expression*⟩ '`<`' ⟨*Expression*⟩
| ⟨*Expression*⟩ '`>=`' ⟨*Expression*⟩
| ⟨*Expression*⟩ '`>`' ⟨*Expression*⟩
| ⟨*Expression*⟩ '`+`' ⟨*Expression*⟩
| ⟨*Expression*⟩ '`-`' ⟨*Expression*⟩
| ⟨*Expression*⟩ '`*`' ⟨*Expression*⟩
| ⟨*Expression*⟩ '`/`' ⟨*Expression*⟩
| ⟨*Expression*⟩ '`%`' ⟨*Expression*⟩
| ⟨*Expression*⟩ '`&`' ⟨*Expression*⟩
| ⟨*Expression*⟩ '`|`' ⟨*Expression*⟩
| ⟨*Expression*⟩ '`^`' ⟨*Expression*⟩
| '`!`' ⟨*Expression*⟩
| '`-`' ⟨*Expression*⟩
| '`~`' ⟨*Expression*⟩
| '`&`' ⟨*Expression*⟩
| '`*`' ⟨*Expression*⟩
| '`(`' ⟨*Typ*⟩ '`)`' ⟨*Expression*⟩
| 'length' ⟨*Expression*⟩
| '`new`' ⟨*Type*⟩ ('`[`' ⟨*Expression*⟩ '`]`')?
| '`delete`' ('`[]`')? ⟨*Expression*⟩
| ⟨*Expression*⟩ '`[`' ⟨*Expression*⟩ '`]`'

Figure 6: Overview of expression operators in IVL. The semantic corresponds to the C++ operators with the same name.

⟨*Atom*⟩ ::= ⟨*Number*⟩
| ⟨*Identifier*⟩
| ⟨*Character*⟩
| ⟨*String*⟩
| ⟨*Identifier*⟩ '`(`' ⟨*Expression*⟩? ('`,`' ⟨*Expression*⟩)* '`)`'
| '`false`'
| '`true`'
| '`?`' '`<`' ⟨*Type*⟩ '`>`'
| '`?`' '`(`' ⟨*Type*⟩ '`)`'
| '`@result`'
| '`(`' ⟨*Expression*⟩ '`)`'

Figure 7: Overview of atomic expressions. the lexical rules for identifiers, strings and characters are similar to those of C++.

a valid expression. An overview of the existing atomic expressions is shown in Fig. 7, the existing operators are listed in Fig. 6.

Both the semantics and the priority of the operators correspond to the same named operators of the C ++ language. The *length* operator has no equivalent in C++. It is only defined on arrays and as a result it returns the length of the given array. Pointer operations including pointer arithmetic, index access and dereferencing are supported as well.

To define a symbolic expression, an underlying primitive data type needs to be specified. The symbolic expression can take any value from the range of values of the underlying data type. During execution of the program, this value range can be additionally restricted. This can be done implicitly (for example, by a conditional jump) or explicitly via the *assume* statement. Fig. 8 shows an example of using symbolic expressions.

Each expression is associated with three basic properties:

- Every expression has a type;
- A property that indicates if the expression be used on the left side of an assignment [1] (lValue);

[1] An assignment can be a direct assignment, a variable initialization or a parameter transfer during a function call

Table 2: Problems with nested blocking expressions

Compound expressions may block due to nested function calls. This can lead to a more complex back-end implementation, since a single *IP* (instruction pointer) at the statement level would be insufficient. For example, suppose a thread contains a statement *int x = f() + g()*. So on the right side of the assignment is a nested expression. If function call f is blocking, then the underlying thread would also be blocked. However, resuming execution is no longer possible with a single *IP* because the *IP* always points to a statement (rather than a subexpression within the statement). However, the last executed statement was not fully completed because the nested expression blocked. However, it is not saved how much of the expression has already been processed and, above all, the results of the subexpressions are lost when blocking. To deal with this limitation, a new expression has been added, *@result*. This expression refers to the result of the last function call. By introducing new local variables in combination with the *@result* expression, nested blocking function calls can be unpacked, as shown in the following listing. This transformation can be automatically performed by a front-end.

```
1  /* Assuming any of f or g can block (i.e.
      perform a context switch), then the
      expression execution will be
      interrupted */
2  int x = f() + g()
3
4  /* The above statement can be re-written
      as follows (which simplifies context
      switch implementation) */
5  f()
6  int a = @result
7  g()
8  int b = @result
9  int x = a + b
```

Listing 1: Unpacking of nested blocking functions calls

- If the execution of the expression can block (i.e. lead to a context switch due to function calls);

The type of an atomic expression is either predefined (e.g. the expression *true* has the type *bool*) or can be determined directly without further dependencies (e.g. the type of a number depends on the value range). The types of compound expressions are formed using the operator rules. The exact formation rules for the types are based on the type rules of the C++ language and can therefore be looked up in the C++ standard [1].

Only references to memory areas are allowed on the left side of an assignment (lValue). These include variables (local and global), dereferencing pointers, and elements of an array. All other expressions may not appear on the left side of an assignment. The generic term assignment refers to a direct assignment and a variable initialization or a parameter transfer in a function call.

The IVL supports blocking instructions, i.e. context switches due to wait instructions. Expressions can not actually (directly) block. The only expression that can block is the function call, because a function call executes the instructions of the called function, which in turn can have a context switch. If a blocking instruction is executed, then ultimately also the function call is blocked. Table 2 shows an example. The property of whether an expression is blocking can be determined statically. For example by simply assuming that a function with at least one blocking instruc-

```
1  main begin
2    /* Variable x is assigned a symbolic expression of type int */
3    int x = ?<int>
4    if x > 5 goto cont:
5      /* Add constraint to path condition */
6      assume x >= 2
7      /* The conditional goto also led to an extension of the path condition */
8      assert x >= 2 && x <= 5
9      goto done
10   cont:
11   /* is valid due to the conditional jump */
12   assert x > 5
13   done:
14 end
```

Figure 8: Example program that illustrates symbolic expressions

Table 3: Mapping of primitive data types. The bit-witdh can be configured to match the C++ execution semantics of the underlying machine.

Type	Bit-width	C++/SystemC equivalent
char	8	char
short	16	short
int	32	int
long	64	long
uchar	8	unsigned char
ushort	16	unsigned short
uint	32	unsigned int
ulong	64	unsigned long
bool	1	bool
void		void
event		sc_core::sc_event
sc_int	N	SystemC signed bitvector (configurable bit-width)
sc_uint	N	SystemC unsigned bitvector (configurable bit-width)

tion may also be able to block. The function call then takes over the property of the function. Composite expressions inherit the properties of the subexpressions. To avoid dealing with blocking expressions in the back-end, which in turn simplifies the implementation of context switches, we have added the *@result* expression as discussed in the example in Table 2. A front-end can automatically perform the required transformation.

4.4 Type system

The type system can also be defined recursively. The foundation is formed by the basic types. Existing types can be used to define array and pointer types (compound types). Fig. 9 shows an overview of the type system.

The existing primitive data types can be found in Table 3. They are based on the primitive data types of C++. Type deduction and implicit propagation rules are based on the corresponding C++ semantics. The specified bit-width and the resulting range of values are configurable and can be adapted to the underlying machine architecture as necessary to conform to the C++ execution semantics.

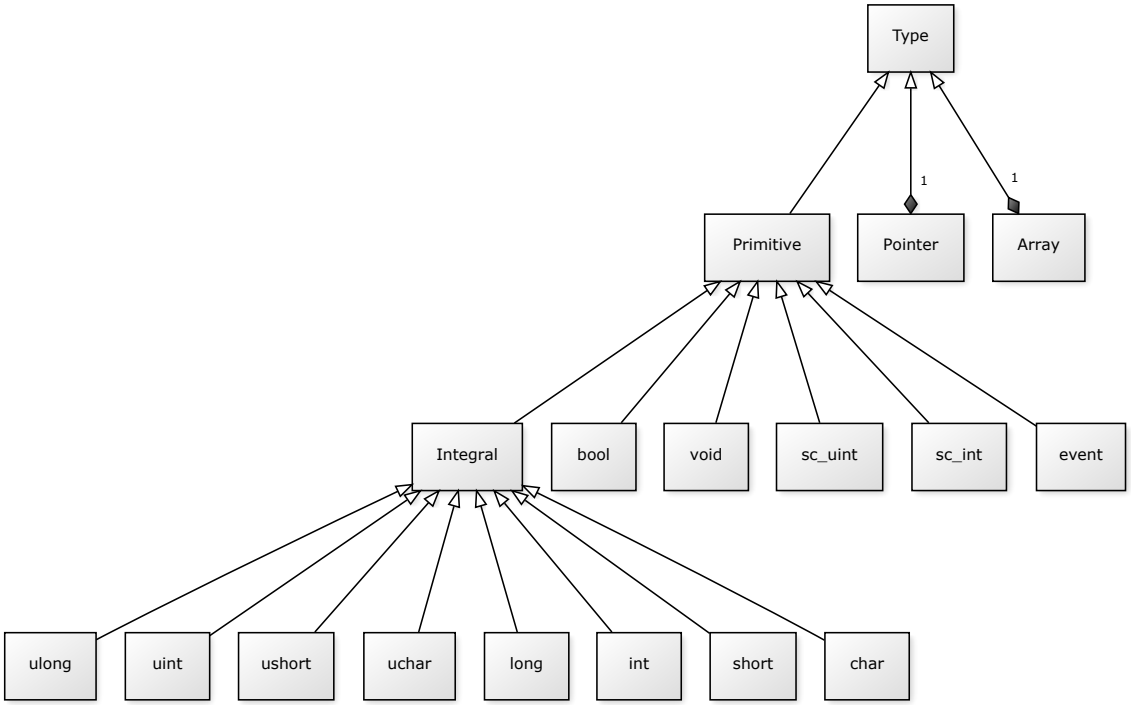


Figure 9: Type system overview

5 Execution Semantics

The simulation semantics of SystemC (see [6]) is precisely followed by the IVL primitives. Essentially, if multiple IVL threads are runnable, one of them will be non-deterministically selected. This thread is then executed non-preemptively until it finishes or suspends itself by calling *wait*. This causes a context switch back to the scheduler, which can again select another runnable thread. If no runnable thread is available, the scheduler performs pending delta or timed notifications accordingly to activate waiting threads. In the following we provide more details on the IVL simulation semantics.

The basis of the simulation phase are the threads. Threads are completely managed by the simulator back-end, they can not be called manually from the program. The execution of a thread is not preemptive, i.e. a running thread is never interrupted by the simulator, it must itself execute a blocking instruction. The intermediate language thus defines cooperative multitasking for executing the threads. Each thread is in one of three states: *runnable*, *blocked*, or *terminated*. A runnable thread can be executed by the simulator. It switches to the state *blocked* when it executes a blocking instruction (waiting for an event or for time or suspends itself) or is suspended by another thread. The blocking is canceled when the expected event occurs. Initially, all threads are runnable following the simulation semantics of SystemC. Threads are only considered for execution once the simulation is started by the *start* function. A thread terminates if all its statements have been executed. The thread state *terminated* can not be left again.

The execution of the IVL program starts with the function *main*. As soon as the *start* statement is executed, execution transfers from the initialization phase to

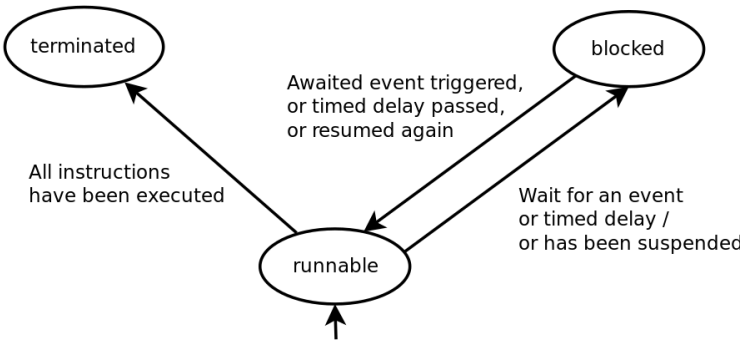


Figure 10: Overview of the thread states

the simulation phase. The function *main* is blocked and the threads are considered for execution. The simulation phase ends once no more thread is runnable, i.e. all of them are blocked or terminated and no more event to unblock any of the threads is pending. The IVL simulation semantics closely follow the simulation semantics of SystemC. For example see [4] for an short overview of the SystemC simulation semantics. Essentially, the simulation phase can be subdivided into four sub-phases:

1. *Evaluation*: This is the initial phase of the *start*-initiated simulation phase. As long as there are executable threads, one of them is selected and executed until it blocks. If there are no more executable threads, the simulation goes into phase two.

2. *Update*: The update functions are executed. These updates have been requested during the evaluation phase (or the startup phase, if the *update* phase is executed once as part of the initialization phase). The evaluation phase together with the update phase corresponds to a *delta cycle* of the simulation.

3. *Delta-Notify*: In this phase, all delta notifications are performed. This will notify the associated events. This causes threads that were blocked on these events to return to state *runnable*. If there is at least one runnable thread at the end of this phase, then the simulation goes into phase one, otherwise in phase three.

4. *Advance-Time*: This phase increases the current simulation time. For this purpose, the minimum time of all delayed events and time-blocked threads is computed. The current simulation time is incremented by the calculated value. If the simulation exceeds the optionally specified maximum time, then the simulation is finished. All threads who finished their waiting time are changed to state *runnable* and all events whose delay has expired are notified, which in turn can unblock waiting threads. If the number of runnable threads is empty at the end of this phase, then the simulation phase is terminated, otherwise it goes into phase one.

As soon as the simulation phase is ended, the execution of the program enters the final cleanup phase. All remaining statements of *main* will be executed here. Then the program execution is finished.

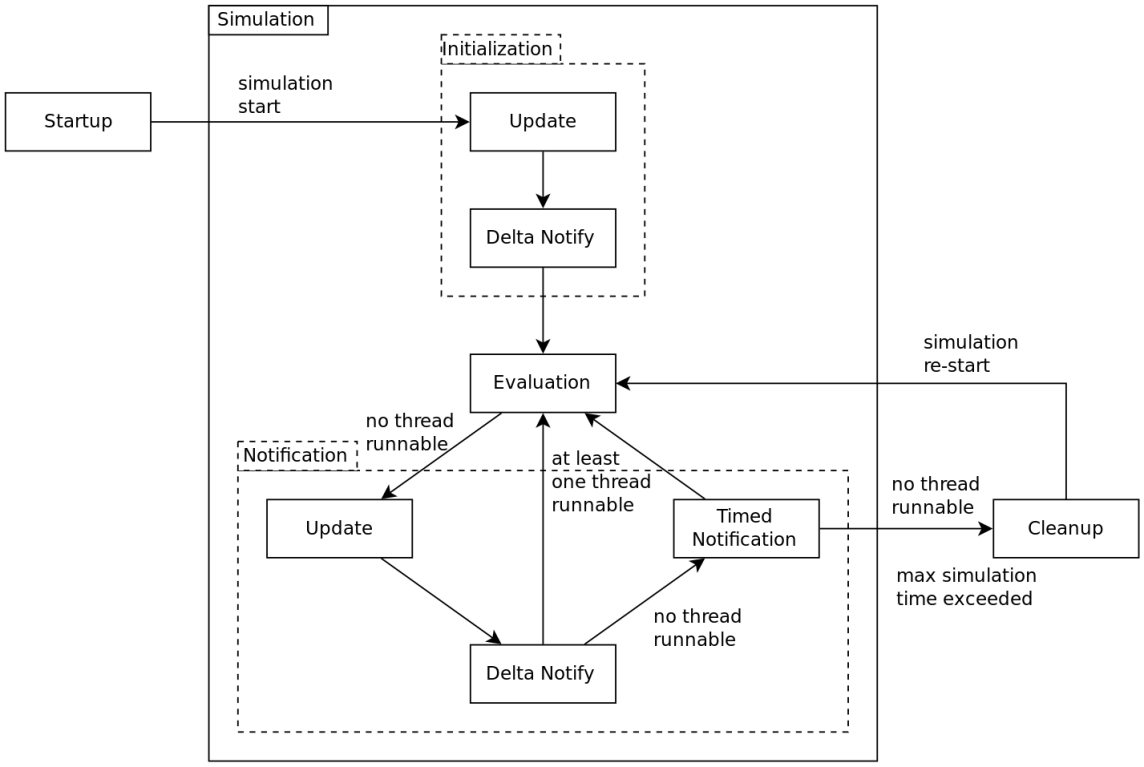


Figure 11: Simulation starts with the initialization phase and performs one or more cycles of evaluation, update and delta-/timed-notifications until the maximum simulation time is reached or no more thread is runnable after performing notifications.

6 Example Translation: SystemC FIFO in IVL

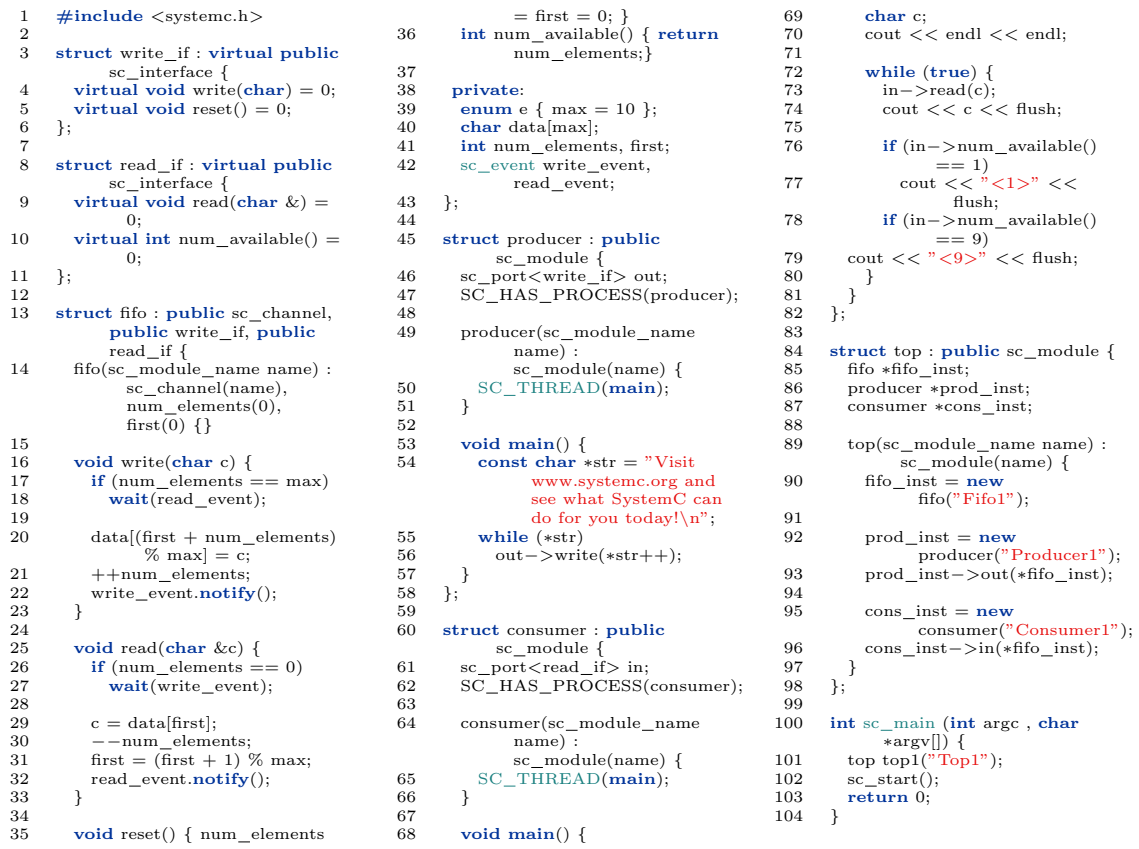
A SystemC design has first to be translated into an IVL program before formal verification is applied. For illustration, in this section we translate an example SystemC program into the IVL.

The chosen SystemC example program is part of the SystemC distribution. It is located in the examples folder under the name simple_fifo. Fig. 12 shows the program in a slightly shortened form. The cuts are exclusively syntactic in nature, both programs are semantically equivalent. The corresponding translated version in IVL is shown in Fig. 13.

6.1 FIFO Behavior Description

Essentially, the program consists of three components: a producer (*producer*), a consumer (*consumer*) and a fifo. The producer produces individual characters and writes them into the fifo. The consumer removes them from the fifo and outputs them on the standard output. The fifo represents a communication channel between producer and consumer. It can only hold *max* many characters, then it is full. Producer and consumer thus alternate their access if the fifo is full or empty.

The fifo provides a total of four operations. The functions *read* and *write* a take or add a character, respectively. The *num_available* function specifies how many characters are currently present. A call to *reset* resets the fifo to the initial state.



```
#include <systemc.h>

struct write_if : virtual public sc_interface {
  virtual void write(char) = 0;
  virtual void reset() = 0;
};

struct read_if : virtual public sc_interface {
  virtual void read(char &) = 0;
  virtual int num_available() = 0;
};

struct fifo : public sc_channel, public write_if, public read_if {
  fifo(sc_module_name name) : sc_channel(name), num_elements(0), first(0) {}

  void write(char c) {
    if (num_elements == max)
      wait(read_event);

    data[(first + num_elements) % max] = c;
    ++num_elements;
    write_event.notify();
  }

  void read(char &c) {
    if (num_elements == 0)
      wait(write_event);

    c = data[first];
    --num_elements;
    first = (first + 1) % max;
    read_event.notify();
  }

  void reset() { num_elements = first = 0; }
  int num_available() { return num_elements;}

 private:
  enum e { max = 10 };
  char data[max];
  int num_elements, first;
  sc_event write_event, read_event;
};

struct producer : public sc_module {
  sc_port<write_if> out;
  SC_HAS_PROCESS(producer);

  producer(sc_module_name name) : sc_module(name) {
    SC_THREAD(main);
  }

  void main() {
    const char *str = "Visit www.systemc.org and see what SystemC can do for you today!\n";
    while (*str)
      out->write(*str++);
  }
};

struct consumer : public sc_module {
  sc_port<read_if> in;
  SC_HAS_PROCESS(consumer);

  consumer(sc_module_name name) : sc_module(name) {
    SC_THREAD(main);
  }

  void main() {
    char c;
    cout << endl << endl;

    while (true) {
      in->read(c);
      cout << c << flush;

      if (in->num_available() == 1)
        cout << "<1>" << flush;
      if (in->num_available() == 9)
        cout << "<9>" << flush;
    }
  }
};

struct top : public sc_module {
  fifo *fifo_inst;
  producer *prod_inst;
  consumer *cons_inst;

  top(sc_module_name name) : sc_module(name) {
    fifo_inst = new fifo("Fifo1");

    prod_inst = new producer("Producer1");
    prod_inst->out(*fifo_inst);

    cons_inst = new consumer("Consumer1");
    cons_inst->in(*fifo_inst);
  }
};

int sc_main (int argc , char *argv[]) {
  top top1("Top1");
  sc_start();
  return 0;
}
```

Figure 12: Fifo in SystemC: Is semantically equivalent to Fig. 13

Initially, the fifo is empty, so it can still hold *max* many characters.

The function *read* blocks the underlying thread on the event *write_event* if the fifo is empty. Likewise, *write* blocks on the event *read_event* if the fifo is full. If a character is successfully read or written, the associated event is notified and any thread blocked on it can be run again.

Producer and consumer are realized as threads. Initially, they are both executable. They are executed in alternating order because thread execution is not preemptive (cooperative multitasking is used).

The producer notifies the consumer, writes *max* many characters into the fifo and then executes the blocking instruction *wait write_event*. Now the producer is blocked and the consumer is able to run. Then[2] the consumer notifies the producer, removes all *max* characters from the fifo and blocks with the execution of *wait read_event*. Now the consumer is blocked and the producer is able to run again.

As soon as the complete string has been written, the creator thread exits. The consumer removes the last produced characters and is blocked himself, since the fifo is now empty. Thus, all (both) threads are blocked, so there is no more executable thread, so the simulation execution is finished.

[2]If the execution would start with the consumer, then he would block himself, because initially the fifo is empty. Then the producer is called and notifies the consumer, etc.

```
1  //////// FIFO BEGIN //////
2  uint max = 10
3  char data[max]
4  int num_elements = 0
5  int first = 0
6  event write_event
7  event read_event
8
9  void write(char c) begin
10   if num_elements != max goto
           cont
11     wait read_event
12   cont:
13
14   data[(first + num_elements)
           % max] = c
15   num_elements =
           num_elements + 1
16   notify write_event
17 end
18
19 void read(char *c) begin
20   if num_elements != 0 goto
           cont
21     wait write_event
22   cont:
23
24   *c = data[first]
25   num_elements =
           num_elements - 1
26   first = (first + 1) % max
27   notify read_event
28 end
29
30 void reset() begin
31   first = 0
32   num_elements = 0
33 end
34
35 int num_available() begin
36   num_elements
37 end
38 //////// FIFO END //////
39
40 thread consumer begin
41   char c
42   loop:
43     read(&c)
44     print c
45     if num_available() != 1
           goto cont1
46       puts "<1>"
47     cont1:
48     if num_available() != 9
           goto cont2
49       puts "<9>"
50     cont2:
51   goto loop
52 end
53
54 thread producer begin
55   char *str = "Visit
           www.systemc.org and
           see what SystemC can
           do for you today!\n"
56   uint len = length(str) - 1 /*
           strings are zero
           terminated */
57
58   uint idx = 0
59   loop:
60     write(str[idx])
61     idx = idx + 1
62     if idx < len goto loop
63 end
64
65
66 main begin
67   /* simulation stops once no
           more thread is runnable
           */
68   start
69 end
```

Figure 13: Fifo in IVL: Is semantically equivalent to Fig. 12

6.2 Translation to IVL

The architecture of SystemC program is created at runtime (i.e. dynamically) by executing the elaboration phase. Modules are instantiated, bindings performed and the individual threads are registered at the simulation kernel. Then the simulation phase is started. The elaboration phase step is is omitted for the IVL. Here the architecture is already statically provided.

Thus, when translating a SystemC program, it must be determined which threads are available in the SystemC design in order to map them to IVL threads. In this example, two threads are used, one for the producer and the consumer. The fifo does not use threads, it is only used for communication between the creator thread and the consumer thread, but only uses normal functions. All three components are realized as classes. Classes are not supported by the IVL. Accordingly, all attributes (methods and variables) of a class are unpacked during the translation step. A unique name prefix might be added in this translation step to avoid name clashes due to overloaded functions and mapping all functions into the global namespace. In this case, it is not necessary because the names are also unique outside the class. A class method can be mapped to either a function or a thread. In this case, producers and consumers are mapped to threads and the fifo to normal functions.

The instantiation of the top-level module and the binding of ports to interfaces is omitted in the IVL, because the assignments are already given by the static structure of the program, i.e. the architecture obtained from a SystemC front-end.

The individual functions (as well as threads [3]) can be straightforwardly translated because the syntax (and also semantics) of the IVL operations is based on C++. Mainly it is required to map loops and conditional statements are to a combination of goto and labels, and also map event handling functions are to the keywords provided. This is necessary because events in SystemC are implemented

[3]Similar to functions, threads also contain a list of instructions. However, they have a simplified prototype, since they do not take any arguments.

as structures (structures are not supported by the intermediate language).

7 References

[1] ANSI and ISO. C++ Standard - ANSI ISO IEC 14882 2003, 2003.

[2] D. Black, J. Donovan, B. Bunton, and A. Keist. *SystemC: From the Ground Up*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2nd edition, 2009.

[3] D. Große and R. Drechsler. *Quality-Driven SystemC Design*. Springer, 2010.

[4] D. Große, H. Le, and R. Drechsler. Proving transaction and system-level properties of untimed systemc tlm designs. In *MEMOCODE*, pages 113–122, 2010.

[5] T. Grotker. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[6] IEEE. *IEEE Standard SystemC Language Reference Manual*. IEEE Std. 1666, 2011.

[7] Hoang M. Le, Daniel Große, Vladimir Herdt, and Rolf Drechsler. Verifying systemc using an intermediate verification language and symbolic simulation. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 116:1–116:6, New York, NY, USA, 2013. ACM.