TSL2 Reference Manual

Leonid Ryzhyk

May 11, 2015

Contents

Chapter 1

Introduction

This document describes syntax and semantics of the Termite Specification Language, Version 2 (TSL2). Information about the Termite synthesis tool can be found on the project website `http://termite2.org`. See [1] for a high-level overview of the project.

The rest of the manual consists of two parts. Chapter 2 presents the main TSL2 concepts and informally describes their semantics. Chapter 3 gives a more formal presentation of TSL2 syntax.

Chapter 2

Overview

2.1 Static and dynamic namespaces

TSL2 supports two namespaces: the *static namespace* and the *dynamic namespace*. The static namespace is populated with compile-time objects: *types* and *constants*. The dynamic namespace is populated with runtime objects: *processes*, *variables*, *methods*, and *wires*. Static objects are uniquely identified by their name and syntactic scope. In contrast, runtime objects can be instantiated multiple times within the specification and hence must be referred to relative to their runtime scope.

2.2 Templates

Templates are the principal mechanism for managing both static and dynamic namespaces. A template models a hardware of software entity, such as a device, an OS, or a device driver. It declares a set of static objects (types and constants) and a set of runtime objects. Static objects declared inside a template can be referenced from any part of the specification via the template name. Runtime objects are instantiated together with the template and can be accessed via a reference to a template *instance*.

The following template declares type `word` (static object) and variable `x` (runtime object):

```
template A
  // type declaration
  typedef uint<16> word;
  // variable declaration
  export word x;
endtemplate
```

The `word` type is globally visible via the `::`-notation as `A::word`. It can be used even if template `A` is never instantiated. In contrast, variable `x` can only

be accessed via an instance of `A`.

Templates are instantiated inside other templates. The sole exception is the `main` template, which is implicitly instantiated in the top-level scope. Every complete TSL2 specification must contain a template called `main`.

In the following example, the `main` template creates an instance of `A`, making its variables accessible from `main` via instance name:

```
template main
  instance A a;

  process pmain {
    // assigning variable x of template instance a.
    a.x = 16'd0;
  }
endtemplate
```

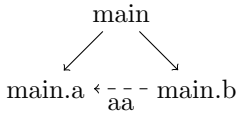
The template instantiation mechanism gives rise to an *instance tree* with the `main` template at its root. Dynamic objects within the tree are accessed using hierarchical identifiers such as `a.x`. This mechanism allows accessing objects down the branch of the instance tree, starting at the local template. In practice, it is often necessary to access objects instantiated in other parts of the tree. This is achieved in a structured way using *template ports*.

A template port is an alias to a template of a given type bound at the time of instantiation. For example, template `B` below declares port `aa` of type `A`, which makes runtime objects in the scope of `A` visible from `B`. Both templates are then instantiated in the `main` template, with the instance of `A` connected to port `aa` of `B`.

```
// template B with port aa of type A
template B(A aa)
  process proc {
    aa.x = 16'd0;
  };
endtemplate

template main
  // create instances of A and B; connect port aa of
  // B to the instance of A.
  instance A a;
  instance B b(a);
endtemplate
```

The following diagram illustrates the resulting instance tree and the link between different branches of the tree via port `aa`.



Another mechanism for managing name spaces is *template inheritance*. Using inheritance, one can create generic templates that capture common properties of a family of entities, leaving some of the properties underspecified. The generic template can be specialised by a child template that fills in the missing details. For example, the following template models common device-class callbacks that must be implemented by any OS specification for the IDE device class. Note that callbacks are defined without bodies, as the exact behaviour is OS-specific.

```
template ide_os
  procedure void write_sectors(uint<48> lba,
    uint<16> sectors, uint<32> buf, bool xfer_error);
  procedure void read_sectors(uint<48> lba,
    uint<16> sectors, uint<32> buf, bool xfer_error);
  procedure void reset();
  ...
endtemplate
```

This template is specialised by the `l4_ide_os` template, that describes the IDE driver interface defined by the seL4 OS.

```
template l4_ide_os(l4_ide_drv drv)
  // the derive statement is used to establish the
  // inheritance relation
  derive ide_os;

  // additional specification items can be declared in the
  // child template
  export iostatus reset_status = ionone;

  // the child template implements methods inherited from
  // the parent.
  procedure void write_sectors(uint<48> lba,
    uint<16> sectors, uint<32> buf, bool xfer_error)
  {
    assert (lba == r_lba);
    ...
  }
```

A template that is missing a method body or a wire assignment (Section 2.4) is called *pure template*. Pure templates can be used in derive statements and in port declarations, however they cannot be instantiated.

Template inheritance is subject to the following rules:

- Derived template must re-declare all ports of all its parent templates with the same names and types. It can also declare additional ports not found in any of its parents.
- If a template has multiple parent templates, then the namespaces of parent templates (consisting of variable, wire, method, goal, and process names)

must not overlap.

- If the derived template re-declares a wire or method declared in one of its parents, the new declaration must have the same signature as the parent declaration.
- The child template can override wire and method declarations of its parent templates. Other parts of the parent template, including variable, goal, and process declarations, are inherited by the child template and cannot be overriden.

2.3 Execution model

All state transitions in TSL2 occur in the context of *processes*. Multiple processes can be enabled at the same time; however exactly one process participates in each individual transition. Process transitions are atomic with respect to other processes. Process that performs the next transition is chosen by the scheduler among enabled processes. Scheduling is fair, i.e., if a process stays enabled sufficiently long, it will eventually get scheduled.

Processes are declared inside templates using the `process` keyword. Such processes become runnable in the initial state of the system. Additional processes can be spawned at runtime using the `fork` construct. TSL2does not allow recursion, hence only a bounded number of processes can be spawned at any time.

In the following example, template `A` contains a single static process `psndrcv`, which spawns two subprocesses `psend` and `preceive`.

```
template A

process psndrcv {
  fork {
    psend:    forever send();
    preceive: forever receive();
  };
  shutdown();
};
...

endtemplate
```

A process state transition starts in the current program location and stops at the next *pause location*. Pause locations can be explicit or implicit. Explicit pause locations are created using `pause`, `wait`, and `stop` constructs. Implicit pause locations are introduced automatically by the compiler in the following cases:

- Before `fork` statements

- Before magic blocks 2.6
- On exit from controllable tasks (see below)

A process can invoke *methods* declared inside its own template as well as in other template instances, via hierarchical identifiers discussed in the previous section. There are three types of methods in TSL2: *functions*, *procedures*, and *tasks*. Functions and procedures must complete instantaneously, i.e., they are not allowed to contain explicit or implicit pauses. In addition, functions are not allowed to have side effects, i.e., they cannot modify any global variables, perform pointer dereferences, or contain assertions (pointer dereferences and assertions statements can have the side effect of taking the system into an error state).

Tasks can consist of multiple atomic transitions. A task can have an optional `controllable` or `uncontrollable` qualifier. Controllable tasks are the only kind of task that can be invoked from synthesised code; although they can also be called from manually written code. They are used to model the device register interface and OS callbacks. In the current implementation of the TSL2 compiler, controllable tasks are subject to additional constraint: they are not allowed to contain internal pause locations, i.e., a controllable task must always complete in a single transition.

Uncontrollable tasks represent driver methods invoked by the OS or the device. Uncontrollable tasks are the only kind of task that can contain automatically generated code, i.e., a magic block can only be placed inside an uncontrollable task.

Controllable task invocations introduce implicit pause locations on return from the task. In the following example execution of process `p1` that invokes controllable task `t1` consists of two transitions.

```
template A
  uint<16> x;
  uint<16> y;

  process p1 {
    x = t1();
    y = x
  };

  task uncontrollable void t1(uint<16> arg) {};
endtemplate
```

The first transition controllable task `t1` and stores its retusr value in variable `x`. An implicit pause is inserted at the task return location. The remainder of the process is executed in the second transition. Note that other processes can run and potentially modify the value of `x` in between transitions 1 and 2.

Tasks without `controllable` and `uncontrollable` qualifiers behave as if the body of the task was inlined at the call location. No additional pauses are inserted before or after the task.

2.4 Variables and wires

Variables are used to store state that persists for the lifetime of the variable. Variables in TSL2 can be declared in the template, process, or method scope. Template-scope variables, also called *global variables*, are instantiated together with the template and are visible from anywhere inside the template. In addition, variables declared with the `export` qualifier can be accessed from other templates via their hierarchical identifiers. Process and method variables are only visible within the syntactic scope of the process or method where they are declared.

In contrast to variables, *wires* are simply aliases to expressions defined over global variables and are not allocated their own storage. Wires keep their values throughout a transition and are updated at the end of the transition. Initial value of a wire is computed based on initial values of global variables.

```
template A
  uint<16> x = 0;
  wire uint<16> w = x + 1;
endtemplate
```

2.5 Correctness specifications

TSL2 provides two mechanisms for specifying correctness conditions on system behaviour:

- *Goals.* A goal is a side-effect-free boolean expression over global variables that must hold infinitely often in any infinite run of the system. A template can declare any number of goals. In addition, TSL2 defines an implicit goal, which requires the system to be outside of a magic block infinitely often; in other words, the system cannot stay inside a magic block forever.

- *Assertions.* An `assert` statement can be placed anywhere inside processes, tasks, and procedures. It defines a condition whose violation immediately transitions the system to an error state.

The following example illustrates the use of goals and assertions in the IDE OS template:

```
template l4_ide_os(ide_drv drv, ide_dev dev)

// The driver/OS interface must infinitely often be in
// a state with no outstanding I/O requests.
goal drv_goal = (req_status == ionone);

task controllable void cache_ack(){
    assert ((req_status == iosuccess_cache) ||
            (req_status == ioerror_cache));
```

```
    req_status = ionone;
};
endtemplate
```

2.6 Magic blocks

A *magic block* is a place holder for automatically generated code. Given a TSL2 specification with one or more magic block, Termite aims to replace magic blocks with executable code so that the resulting complete system satisfies all its goals and does not violate any assertions.

```
template main
  task uncontrollable bool probe() {
    // Magic block
    ...;
    assert ((os.reset_status == iosuccess) ||
            (os.reset_status == ioerror));
    if (os.reset_status == iosuccess) {
      return true;
    } else {
      return false;
    };
};
endtemplate
```

Chapter 3

Syntax reference

3.1 Literals

TSL2 supports boolean literals "`true`" and "`false`" and Verilog-style binary, octal, decimal and hexadecimal integer literals. The exact number of bits can be specified for each integer literal.

```
<intLit> := <decNumber>
          | [<width>] "'b"  <binNumber>
          | [<width>] "'sb" <binNumber>
          | [<width>] "'o"  <octNumber>
          | [<width>] "'so" <octNumber>
          | [<width>] "'d"  <decNumber>
          | [<width>] "'sd" <signedDecNumber>
          | [<width>] "'h"  <hexNumber>
          | [<width>] "'sh" <hexNumber>
<width> := <decNumber>
```

Examples of integer literals:

```
uint<8> x;
sint<8> y;

x = 255;
x = 8'd35;
x = 8'b01010101;
x = 8'b1;

y = 8'sb11111111;
y = 8'sd-6;
```

In case literal width is not specified explicitly, the compiler assumes the smallest width sufficient to encode the given integer value, for example literal 5 is assumed to be 3 bits wide, as 3 bits are sufficient to encode values between

0 and 5. The compiler does not perform automatic truncation or extension of integers. For example, the following assignment statement is invalid, as the left and the right-hand sides of assignment have different width:

```
uint<8> x = 0; // error: x has width 8, while
               // literal 0 has width 1

uint<8> y = 8'd0; // ok
```

3.2 Identifiers

Identifiers are used to refer to static and runtime objects (Section 2.1). TSL2 supports three forms of identifiers: *simple identifiers*, *statically scoped identifiers*, and *hierarchical identifiers*.

Simple identifiers. A simple identifier is a name of a static or runtime object visible within the current syntactic scope. This includes objects declared within the current scope or in one of its parent scopes. For example, simple identifier `x` used in a method body can refer to a local variable or argument of the method, template-global variable, or a type or constant declared in the template or top-level scope.

```
<ident> := (<letter> | "_") (<letter> | <digit> | "_")*
```

Statically scoped identifiers. These identifiers refer to static objects declared within template scope. They can be used to refer to static objects declared in templates other than the local template.

```
<staticIdent> := <ident> "::" <ident>
```

where the first identifier is a template name.

Hierarchical identifiers. These identifiers are used to refer to runtime objects outside of the current syntactic scope. A hierarchical identifier is a dot-separated sequence of literals that traverses the instance tree (Section 2.2) via port and instance names:

```
<hierarchicalIdent> := <ident> ["." <ident>]*
```

The following example illustrates the use of different types of identifiers.

```
template A
  typedef uint<16> word;
  export word x;
endtemplate
```

```
template B(A aa)
  // Reference to the word type declared in template A
  // via static scoped identifier;
  A::word y;

  process proc {
    // In the following statement:
    // * y is a simple identifier that refers to a variable
    //   declared in the local template
    // * aa.x is a hierarchical identifier that refers to
    //   variable x declared in an instance of template A
    //   referred to by the aa port of template B.
    y = aa.x;
  };
endtemplate

template main
  // create instances of A and B; connect port aa of
  // B to the instance of A.
  instance A a;
  instance B b(a);
endtemplate
```

3.3 Types

Type expressions can occur in various contexts in a TSL2 specification, including variable, method, wire, type, and constant declarations. The language currently supports arbitrary fixed-width signed and unsigned integers, booleans, enums, structs, arrays, and pointers.

```
<typeSpec>   := ( <sintType>    // signed int
                | <uintType>    // unsigned int
                | <boolType>    // boolean
                | <userType>    // user-defined type name
                | <enumType>    // enumeration
                | <structType>) // struct
                <typeModifier>* // type modifiers

<sintType>   := "sint" "<" <decimalNumber> ">"
<uintType>   := "uint" "<" <decimalNumber> ">"
<boolType>   := "bool"
<userType>   := <staticIdent>
<enumType>   := "enum" "{" (<ident> ",")*  <ident> "}"
<structType> := "struct" "{" (<typeSpec> <ident> ";")+ "}"

// A type modifier is either array dimension or the pointer
// modifier ("*")
<typeModifier> := ("[" <expr> "]")
```

```
                | "*"
```

TSL2 enum's are different from C-style enum's in that they are not integers. In particular, a variable of enum type cannot be cast to an integer. Such a variable can only take one of the values in the enumeration and not any other arbitrary integer value. Furthermore, an enum declaration cannot assign integer values to enumerators.

Type expressions are subject to the following constraints:

- Enumerations can only be declared in `typedef` statements, i.e., they cannot be declared inline in variable or method declarations. This ensure that every enum type has a name.
- Variable-size arrays are not supported: array dimensions must be compile-time constants.

Examples of type expressions:

```
typedef uint<16> t1;
typedef sint<13> t2;
typedef enum {e1, e2, e3} t3;
typedef struct {t1 f1; t2 f2;} t4;
const t1 c1 = 16'd5;
typedef t4[c1] arrtype;
typedef struct {bool f1; uint<16>[10] f2;} * structptr;
```

3.4 Expressions

TSL2 expressions are constructed from identifiers (Section 3.2), literals (Section 3.1), and method invocations 3.5.10 using operators summarised in Table 3.1 in the order of decreasing precedence.

In addition, TSL2 supports three kinds of conditional expressions: `if-else` expressions (or ternary expressions), `case`-expressions, and `cond`-expressions.

```
<ternExp> := "if" <expr> <expr> "else" <expr>
<caseExp> := "case" "(" <expr> ")" "{" (<expr> ":" <expr> ";")*
                 (<expr> ":" <expr> ";")*
                 ["default" ":" <expr> ";"]
           "}"
<condExp> := "cond" "{"
                  (<expr>    ":" <expr> ";")*
                  ["default" ":" <expr> ";"]
             "}"
```

A `case`-expression chooses a value based on the value of its key expression, whereas a `cond`-expression chooses a value to return by evaluating a series of conditions in order. Note that `if-else` expressions are different from `if` statements described in Section 3.5.9.

operator	syntax	arg type	res type	comment
`[:]`	`e[l:h]`	integer	unsigned int	bit slice
`[ ]`	`e[i]`	array	-	array index
`.`	`e.f`	struct	-	struct field
`->`	`e->f`	struct pointer	-	struct dereference
`!`	prefix	bool	bool	boolean negation
`~`	prefix	integer	integer	bit-wise negation
`-`	prefix	integer	integer	unary minus
`*`	prefix	pointer	-	pointer dereference
`&`	prefix	any	pointer	address-of
`==, !=`	infix	any	bool	
`<, <=, >, >=`	infix	integer	bool	
`&`	infix	integer	integer	bit-wise and
`|`	infix	integer	integer	bit-wise or
`^`	infix	integer	integer	bit-wise xor
`++`	infix	integer	integer	bit-vector concatenation
`&&`	infix	bool	bool	boolean and
`||`	infix	bool	bool	boolean or
`=>`	infix	bool	bool	boolean implication
`*`	infix	integer	integer	multiplication
`%`	infix	integer	integer	residue
`+`	infix	integer	integer	plus
`-`	infix	integer	integer	minus

Table 3.1: TSL2 operators

Finally, TSL2 supports struct expressions with explicit or implicit field names:

```
<structExp>   := <staticIdent> "{"
                     (<namedFields> | <anonFields>)
                 "}"
<namedFields> := "." <ident> "=" <expr>
                 [("," "." <ident> "=" <expr>)*]
<anonFields> := <expr>
                 [("," <expr>)*]
```

Note that TSL2 does not provide type casting operations. In particular, it is impossible to convert a pointer to an integer or an integer to a pointer. In addition, the TSL2 compiler enforces the following type and memory safety rules:

- No arithmetic operations are allowed on pointers or enum's
- Bit-wise operators must be applied to integer operands of the same width
- `==` and `!=` operations can only be applied to operators of identical types (including width and signedness)
- Labels in a `case`-expression and conditions in a `cond`-expression must be side-effect free
- All branches of a `case`, `cond`, or `if-else` expression must return values of the same type
- Lower and upper bounds of a bit slice must be constant expressions, lower bound must be less than or equal to the upper bound, and both bounds must be smaller than the width of the argument.
- The address-of operator (`&`) can only be applied to memory variables (see Section 3.5.1).

3.4.1 Non-deterministic expressions

Special expression `*` is used to generate non-deterministic values of arbitrary type. The exact type is derived from the context, as illustrated in this example:

```
x = *;  // Generate random value of the same type as x.
f(*,*); // Generate random values that match argument
        // types of f.
```

Note that `*` cannot be used as an atom in a complex expression, for example, the following is not valid:

```
x = * + 2; // Invalid use of *
```

3.4.2 L-expressions

L-expressions are a subset of TSL2 expressions that can be used as the left-hand side of assignment statements and as output arguments of methods. L-expressions are defined by the following rules:

1. Names of variables visible in the current scope, including global variables, local variables, and function arguments are L-expressions
2. All valid expressions of the form `*e` and `e->f` are L-expressions.
3. If `e` is an L-expression then the following are also L-expressions:
   - `e.f`
   - `e[i]`
   - `e[l:h]`

3.4.3 Expression evaluation

Expression evaluation order matters in cases when expression operands contain method calls, which may have side effects. All expression operands are evaluated before the expression itself is evaluated. In particular, if the expression contains method calls, then all of these calls are performed before the expression is evaluated. ***This description reflects how the compiler works at the moment; however this behaviour is counter-intuitive and should be fixed to be more C-like***.

3.5 Statements

Statements are used to define process behaviour and occur in process and method bodies and prefix-blocks (Section 3.10.3).

```
<statement> := <varDecl>   // local var declaration
             | <sseq>      // sequential block
             | <spar>      // parallel blocks
             | <sforever>  // forever loop
             | <sdo>       // do loop
             | <swhile>    // while loop
             | <sfor>      // for loop
             | <sbreak>    // break out of a loop
             | <schoice>   // nondeterministic choice
             | <spause>    // pause
             | <swait>     // wait on a condition
             | <sstop>     // stop
             | <sassert>   // assertion
             | <sassume>   // assumption
             | <site>      // if-then-else statement
             | <scase>     // case statement
```

```
              | <sinvoke>  // method invocation
              | <sassign>  // variable assignment
              | <sreturn>  // return statement
              | <smagic>   // magic block
```

3.5.1 Variable declarations

Variable declaration statements are used to declare local variables visible within the syntactic scope of a process, method or prefix-block. A local variable is visible everywhere within its scope regardless of the exact location where it has been declared. A variable declaration consists of an optional `mem` qualifier, variable type and name and optional initial assignment:

```
<varDecl> := ["mem"] <typeSpec> <ident> ["=" <expr>]
```

The `mem` qualifier labels the variable as in-memory variable, which allows the address-of operator to be applied to this variable or any of its fields. In addition, in performing pointer analysis of a TSL2 specification, one can assume that a pointer can only point to an in-memory variable of a matching type.

In case variable declaration does not specify an initial value for the variable, the value for the variable is chosen non-deterministically.

3.5.2 Sequential blocks

C-style sequential blocks:

```
<sseq> = "{" (<statement> ";")* "}"
```

3.5.3 Parallel blocks

The `fork` construct is used to spawn several child processes.

```
<spar> := "fork" "{"
               (<ident> ":" <statement> ";")*
          "}"
```

Each child process is assigned a unique label, which helps identify processes during debugging. The spawning process blocks waiting for all forked processes to terminate. Forked processes terminate when all of them have reached a *final* state (i.e., a process cannot terminate without waiting for its siblings to be in final states as well). A process is in a final state when it reaches either the end of its execution (i.e., executes its last instruction) or a `stop` statement (Section 3.5.6).

In the following example two dynamically spawned processes send and receive data in an infinite loop.

```
process psndrcv {
  fork {
    psend:    forever {
                  stop;   // final state
                  send();
              };
    preceive: forever {
                  stop;   // final state
                  receive();
              };
  };
  shutdown();
};
```

At each iteration of the loop, they go through final states marked by the `stop` statements. When both processes are in their respective final states, the entire `fork` block *may* terminate, with the parent process moving on to the next statement; however, it is not required to terminate and can continue executing `forever` loops. The choice between terminating and continuing execution is performed nondeterministically by the environment. In contrast, in the following example, once both processes reach their final control locations, the fork block terminates instantaneously, as neither process can execute any more transitions:

```
process psndrcv {
  fork {
    psend:    send();
    preceive: receive();
  };
  shutdown();
};
```

3.5.4 Loops

In addition to conventional do, while, and for loops, TSL2 supports forever loops, which are equivalent to `while(true)`.

```
<sdo>      := "do" <statement> "while" "(" <expr> ")"
<swhile>   := "while" "(" <expr> ")" <statement>
<sfor>     := "for" "(" [<statement>] ";"
                        <expr> ";"
                        <statement> ")"
                    <statement>
<sforever> := "forever" <statement>
```

At the moment TSL2 does not allow *instantaneous loops*, i.e., every possible path through the loop body must contain an explicit or implicit pause location (Section 2.3). This restriction is enforced by the compiler via a simple static

check. Note that there is no implicit pause before the body of the loop, i.e., execution enters the loop instantaneously and continues until reaching a pause location inside the loop. For example, in the following example, process `foo` executes two transitions: (1) `x=16'd0; (x<1) == true; x=x+16'd1`, and (2) `(x<1) == false; y=16'd0`.

```
process foo {
    x = 16'd0;
    while (x < 1) {
        x = x + 16'd1;
        pause;
    };
    y = 16'd0;
};
```

The `break` statement can be used anywhere inside a loop to transfer control to the first instruction following the body of the innermost loop.

3.5.5 Non-deterministic choice

The `choice` construct allows the environment to non-deterministically choose between two or more actions:

```
<schoice> := "choice" "{"
               (<statement> ";")*
             "}"
```

3.5.6 Pause statements

TSL2 offers three ways to insert an explicit pause location (Section 2.3) in the control flow of a process or method: `wait`, `pause`, and `stop` statements.

```
<swait>  := "wait" "(" <expr> ")"
<spause> := "pause"
<sstop>  := "stop"
```

The `wait` statement inserts a pause location and disables the current process until the wait condition becomes true. If the wait condition stays true for sufficiently long time, the process is guaranteed to eventually leave the pause location. Note that if the wait condition is already true when the process enters the pause location, the current transition terminates anyway.

The `pause` statement is a shortcut for `wait(true)`. `stop` behaves like `pause`, but additionally marks the current process state as final (Section 3.5.3).

3.5.7 Assertions

An assertion behaves as a no-op if its argument evaluates to true, and causes a transition to an error state otherwise.

```
<sassert> := "assert" "(" <expr> ")"
```

The argument of an `assert` statement must be a side-effect-free expression.

3.5.8 Assumptions

```
<sassume> := "assume" "(" <expr> ")"
```

Assumptions are used to constrain possible system behaviours. Similar to assertions, they specify constraints that must hold for any valid execution of the system. However, while an assertion causes transition to an error state if its condition is violated, the `assume` statement prunes all transitions that violate the assumption.

Assumptions are particularly useful in imposing restrictions on randomly generated values, as illustrated by the following example.

```
// assign random values for stopbits and data vars
stopbits  = *;
data      = *;
// only certain combinations of stopbits and data values
// are valid
assume(((stopbits==UART_STOP_BITS_15) && (data==4'd5)) ||
       ((stopbits==UART_STOP_BITS_2) && (data!=4'd5)));
case (data) {
    4'd5:       data_bits = CUART_DATA5;
    4'd6:       data_bits = CUART_DATA6;
    4'd7:       data_bits = CUART_DATA7;
    4'd8:       data_bits = CUART_DATA8;
    default: assume(false); // other values are not allowed
};
```

3.5.9 Conditional statements

TSL2 supports C-style `if-else` statements and `case` statements:

```
<site>  := "if" "(" <expr> ")" <statement>
             ["else" <statement>]
<scase> := "case" "(" <expr> ")" "{"
             (<expr> ":" <statement> ";")*
             ["default" ":" <statement> ";"
           "}"
```

Labels of the `case` statement can be arbitrary side-effect-free deterministic expressions of a matching type. At runtime, the first matching label is selected. At most one branch of a case statement is executed; execution does not fall through to the next label automatically. Unlike in C, the `break` statement cannot be used to break out of a case clause. While `break` is allowed inside the body of a `case`, it has the effect of breaking out of the innermost loop.

3.5.10 Method invocation

Method invocations can occur as atoms in expressions as well as standalone statements. Method name is a hierarchical identifier that refers either to a method declared in the local template or exported from another template.

```
<sinvoke> := <hierarchicalIdent> "(" [<expr> [("," <expr>)*]] ")"
```

Method arguments must match the number and types of formal arguments in the method declaration. Output arguments 3.10.5 must be L-expressions 3.4.2.

3.5.11 Assignment statements

```
<sassign> := <expr> "=" <expr>
```

The left-hand side of an assignment statement must be an L-expression (Section 3.4.2). Types of left- and right-hand side expressions must match, including sign and width for integer types.

3.5.12 Return statements

```
<sreturn> := "return" [<expr>]
```

Return statements are only allowed in method bodies. If the method is a `void` method, return must not have an argument; otherwise it must have an argument whose type matches the return type of the method. Every path through a non-void method must contain a `return` statement. Statements following a `return` statement are ignored and control transfers to the location immediately following the call site.

If the method is a controllable task then an implicit pause (Section 2.3) is introduced at the return location.

3.5.13 Magic blocks

```
<smagic> := "..."
```

See section 2.6.

3.6 TSL2 file structure

A TSL2 file consists of *import* statements, type declarations, constant declarations, and template declarations.

```
<tslFile>  := <specItem>*
<specItem> := <import>
            |  <typeDecl>
            |  <const>
            |  <template>
```

3.7 Import statements

Import statements are used to combine multiple TSL2 files. They are only allowed in the top-level syntactic scope, i.e., they are illegal inside template of type declarations. An import statement consists of the `import` keyword followed by file path in angle brackets:

```
import<ide_dev.tsl>
import<os/ide_tsl2/l4_ide.tsl>
import<../../os/ide_tsl2/ide_class.tsl>
```

The TSL2 compiler appends this path to the current directory and to each import directory, specified via the `-I` command line switch, in order, until a file with this name is found.

3.8 Type declarations

Type declarations can be placed in the top-level scope or in a template scope.

```
<typeDecl> := "typedef" <typeSpec> <ident>
```

3.9 Constant declarations

Constant declarations can be placed in the top-level scope or in a template scope. The value of a constant is an expression that can be evaluated at compile time.

```
<const> := "const" <typeSpec> <ident> "=" <expr>
```

Examples:

```
typedef struct {bool f1; uint<16> f2;} stype;

const bool     b = true;
const uint<16> u = 16'd5;
const stype    s = stype {.f1 = b, .f2 = u};
```

3.10 Templates

Templates must be declared within the top-level scope, i.e., nested template declarations are not allowed. Template declaration has the following syntax:

```
<template> := "template" <ident> [(<portDeclarations>)]
                  (<templateItem> ";")*
              "endtemplate"
```

Here, `<portDeclarations>` is a comma-separated list of port declarations, described in Section 2.2. A template item is one of:

- Derive statement
- Instance declaration
- Type declaration
- Constant declaration
- Global variable declaration
- Init block
- Prefix block
- Process
- Method
- Goal
- Wire declaration

Derive statements and instance declarations were considered in Section 2.2. Type declarations and constant declarations were considered in Sections 3.8 and 3.9. We describe the remaining kinds of template items below.

3.10.1 Global variable declarations

Global variables (Section 2.4) are declared in the template scope. The declaration syntax is the same as for local variables (Section 3.5.1) with the optional `export` qualifier, which indicates that the variable can be accessed from outside the template:

```
<gvarDecl> := ["export"] <varDecl>
```

Similar to local variable declarations, if the declaration does not specify an initial value for the variable, the value for the variable is chosen non-deterministically. The initial value can be further constrained by `init` blocks, as described below.

3.10.2 Init blocks

An `init` block defines a constraint over initial assignment of global veriables.

```
<initBlock> := "init" <expr>
```

The body of an `init` block is a boolean expression over global variables visible from the current template, including variables exported from other templates.

The following example constrains initial values of `config_in_progress` and `sendq_head` variables:

```
template linux_uart_drv(uart_dev dev)
    bool config_in_progress;
    uint<16> sendq_head;
    init (config_in_progress == true) &&
         (sendq_head == 16'd0);
endtemplate
```

Note that the same result can be achieved using initial variable assignments:

```
template linux_uart_drv(uart_dev dev)
    bool config_in_progress = true;
    uint<16> sendq_head = 16'd0;
endtemplate
```

In general, however, `init` blocks are a more general mechanism for constraining initial state, for example the following condition cannot be captured using initial variable assignments:

```
template linux_uart_drv(uart_dev dev)
    init (config_in_progress == true) ||
         (sendq_head == 16'd0);
endtemplate
```

A template can contain multiple `init` blocks. In using `init` blocks, one must make sure that different `init` blocks do not contradict each other and initial variable assignments, leading to an empty initial set, as in the following example:

```
template linux_uart_drv(uart_dev dev)
    bool config_in_progress = true;
    init config_in_progress == false;
endtemplate
```

3.10.3 Prefix blocks

A prefix block is an arbitrary statement that is automatically prepended to all transitions of the system. It is intended as a low-level mechanism that allows implementing certain behaviours that are tricky to achieve without it. It should be used with care and will likely be hidden behind syntactic sugar in the future.

```
<prefix> := "prefix" <statement>
```

3.10.4 Processes

See Section 2.3.

```
<processDecl> := "process" <ident> <statement>
```

3.10.5 Methods

A method declaration consists of

- optional `export` qualifier that indicates whether the method can be invoked from outside its template
- method category specifier that labels the method as function, procedure, or task, and in the last case optionally as a controllable or uncontrollable task (Section 2.3)
- return type or `void` if the method does not return a value
- argument list
- method body

```
<methodDecl> := ["export"]                // Exported method?
                <methCateg>               // Method category
                ("void" | <typeSpec>)        // Return type
                <ident> "("                  // Method name
                    [<arg> ("," <arg>)*]     // Arguments
                ")"
                <statement>               // body

<methCateg> := "function"
             | "procedure"
             | "task" [ "controllable"
                      | "uncontrollable"]

<arg> := ["out"] <typeSpec> <ident>
```

Method arguments are used to pass data both to and from the method. An argument can be declared as an input or an output argument, but not both. Syntactically, input and output arguments are distinguished using the `out` qualifier.

Both input and output arguments can be read and modified in the body of the method; however the initial value of an output argument is non-derministic. The final value of an input argument is discarded, while the final value of an output argument is propagated to the L-expression (Section 3.4.2) passed as the actual argument to the method.

3.10.6 Goals

```
<goalDecl> := "goal" <ident> "=" <expr>
```

See Section 2.5.

3.10.7 Wire declarations

Wire declaration (Section 2.4) consists of wire type, wire name, and optional wire expression.

```
<wire> := ["export"] "wire" <typeSpec> <ident> ["=" <expr>]
```

The wire expression is defined over global variables and wires of the current template and other templates accessible via hierarchical identifiers. However, circular dependencies among wire expressions are forbidden. If the wire expression is omitted, the wire is a *pure wire* and must be re-defined in child templates. In merging template with its parents, the TSL2 compiler picks the last wire declaration in the inheritance hierarchy.

Bibliography

[1] L. Ryzhyk, A. Walker, J. Keys, A. Legg, A. Raghunath, M. Stumm, and M. Vij. User-guided device driver synthesis. In *OSDI*, Broomfield, CO, USA, Oct. 2014.