The NSDL Process Flow

Bradford G. Van Treuren and Michele Portolan

November 2008

Outline

- System Level Embedded Testing
- Working Group History (April 2008 Discussion)
- LabVIEW Model
- Generalized Batch Process Flow
- Generalized Interactive Process Flow
- NSDL Instrument Description Model

Alcatel·Lucent

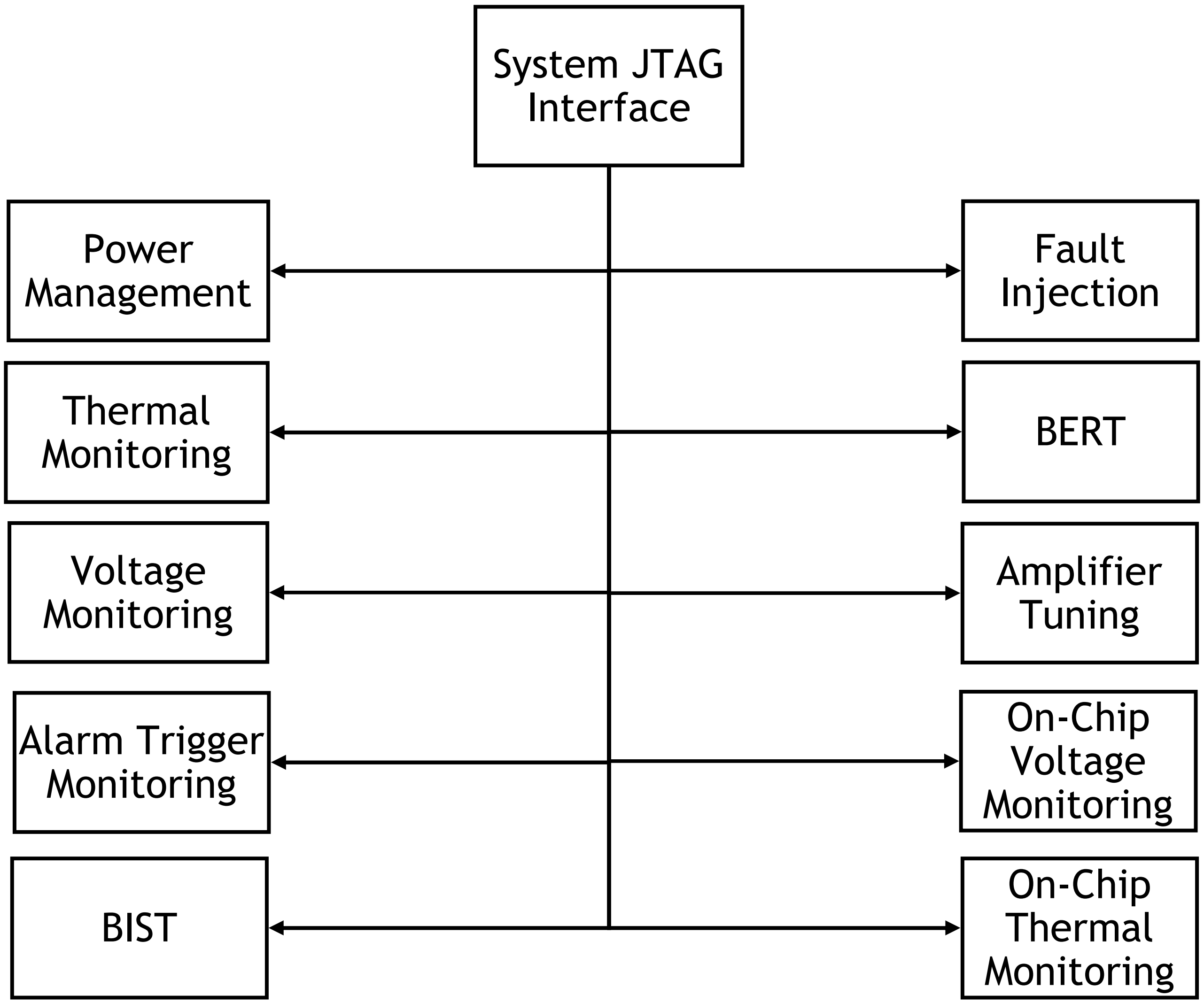
The NSDL Process Flow

System Level Embedded Testing

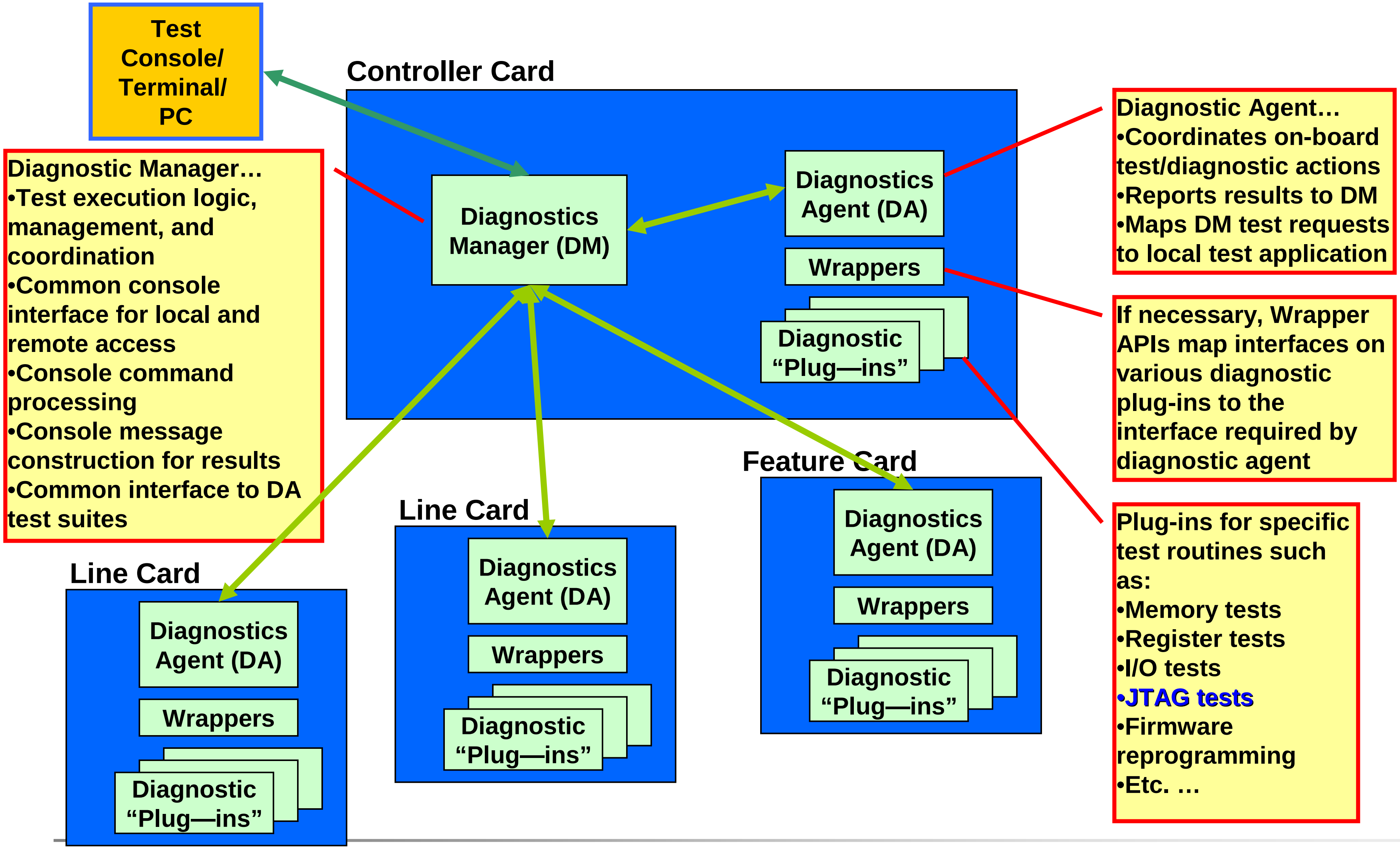
Bradford G. Van Treuren and Michele Portolan

November 2008

Examples of Instrumentation Used at the System Level

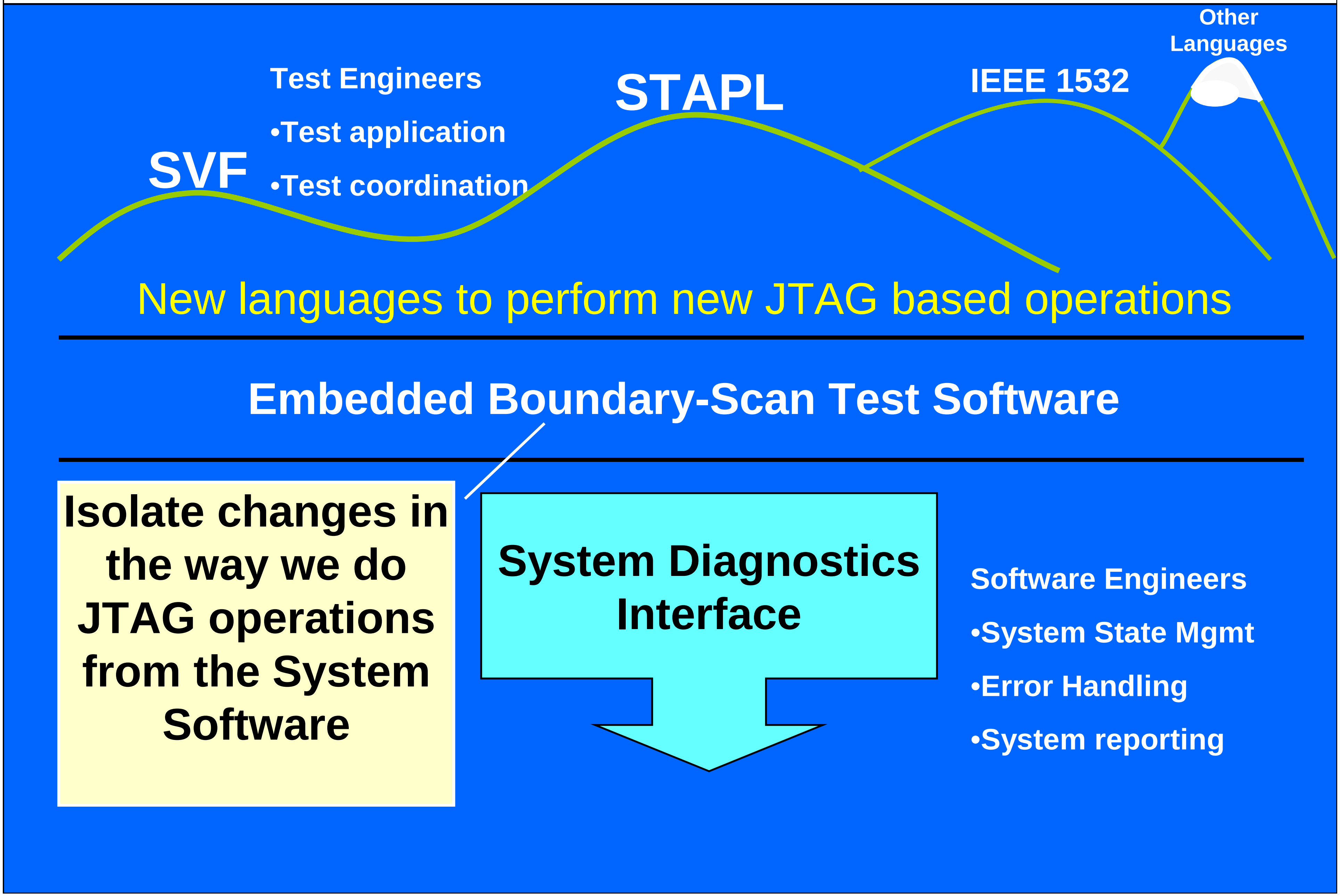


Typical Embedded Systems Application Software



Alcatel·Lucent

System JTAG Integration Role



Alcatel·Lucent

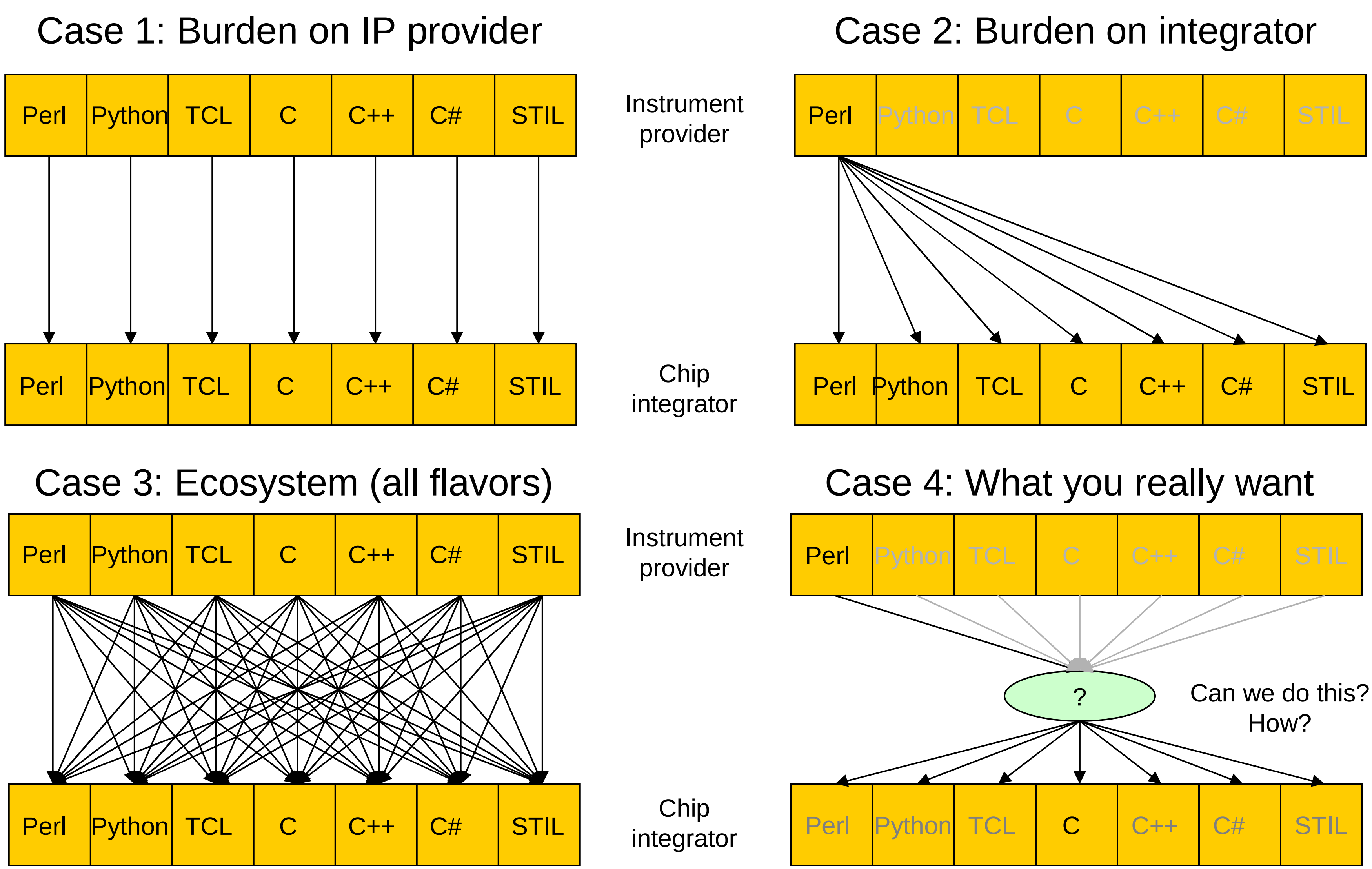
The NSDL Process Flow

Working Group History (April 16, 2008)

Bradford G. Van Treuren and Michele Portolan

November 2008

iJTAG SW problem statement (Jeff-R)

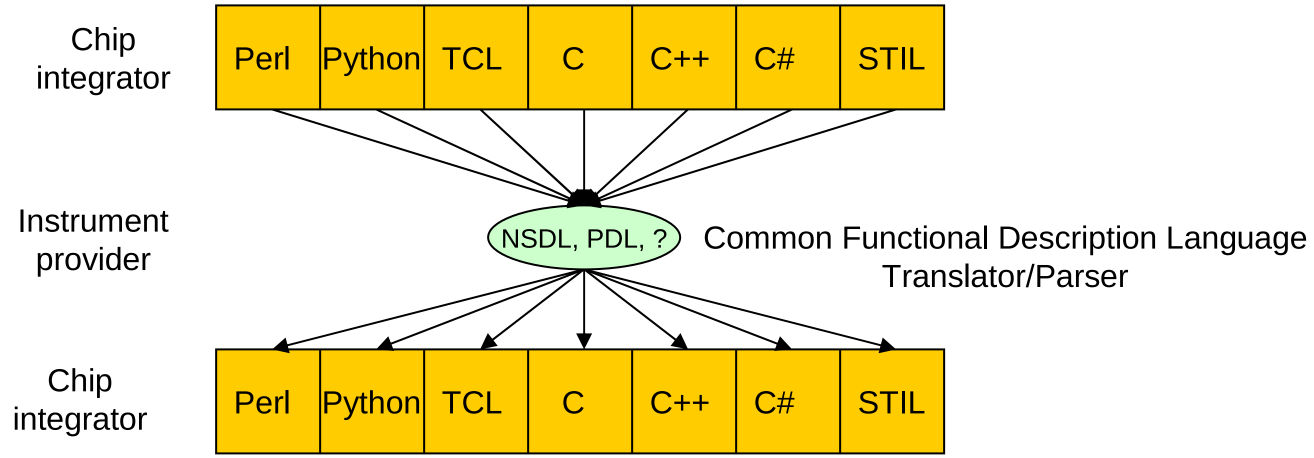


4/16/08

Alcatel·Lucent

iJTAG Software Problem: Common Functional Description Language Translation

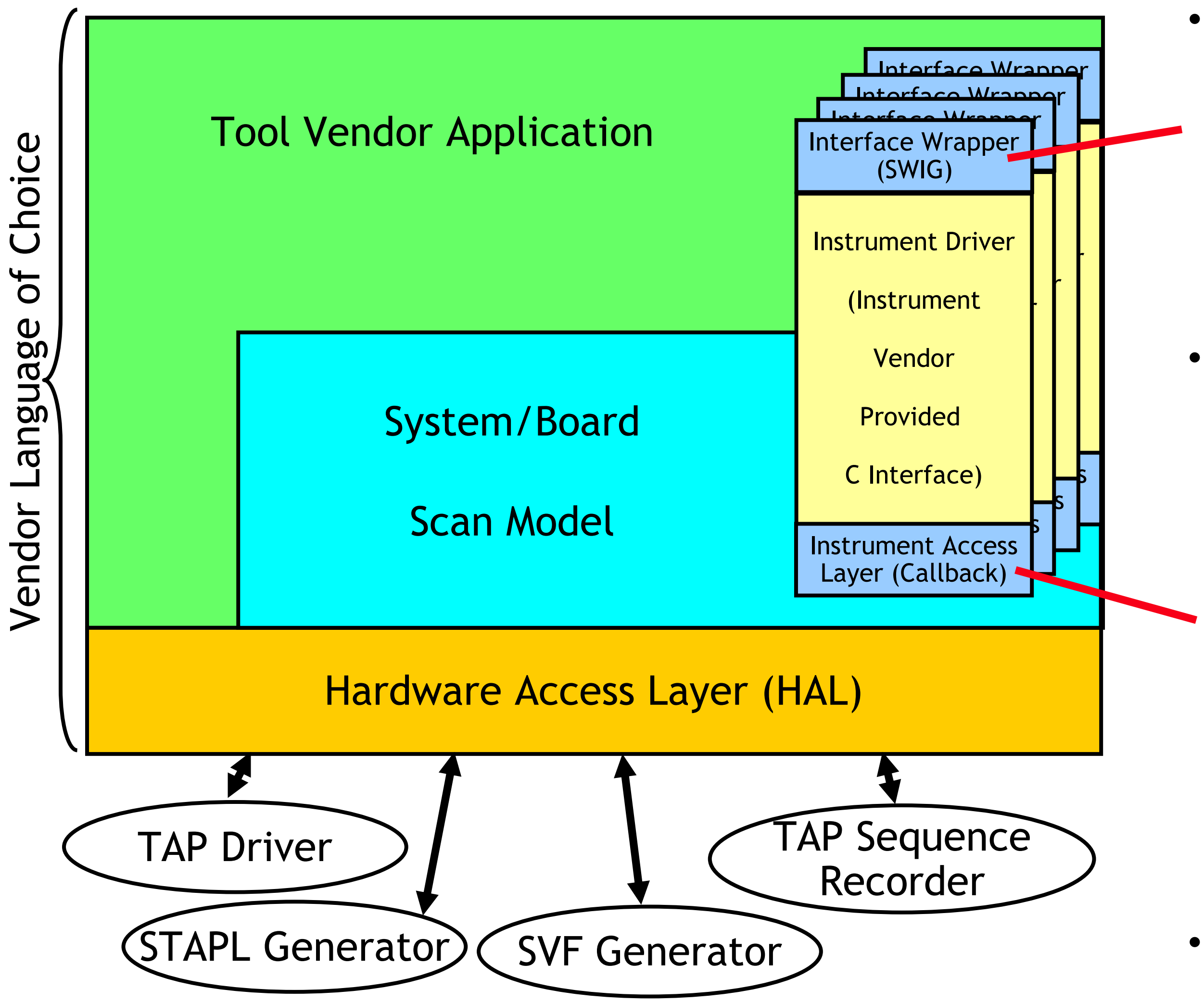
Case 5: Translations from a Common Functional Description Language



- Standardizes language to simplify instrument provisions and interpretation
- Provides for high level problem domain descriptions of functions
- Allows EDA and Scan Tool vendors to implement functions in their own tool environment/architecture instead of a 1687 view architecture
- Supports efficient flow control generation for dynamic control of instruments including support for efficient embedded control (e.g., STAPL control flow)
- Could leverage existing VHDL flow control description to simplify tool integration
- Tool interpretations may differ if language is too ambiguous

Alcatel·Lucent

Typical Scan Tool Vendor Implementation for Case 4 with proposed “C” Driver

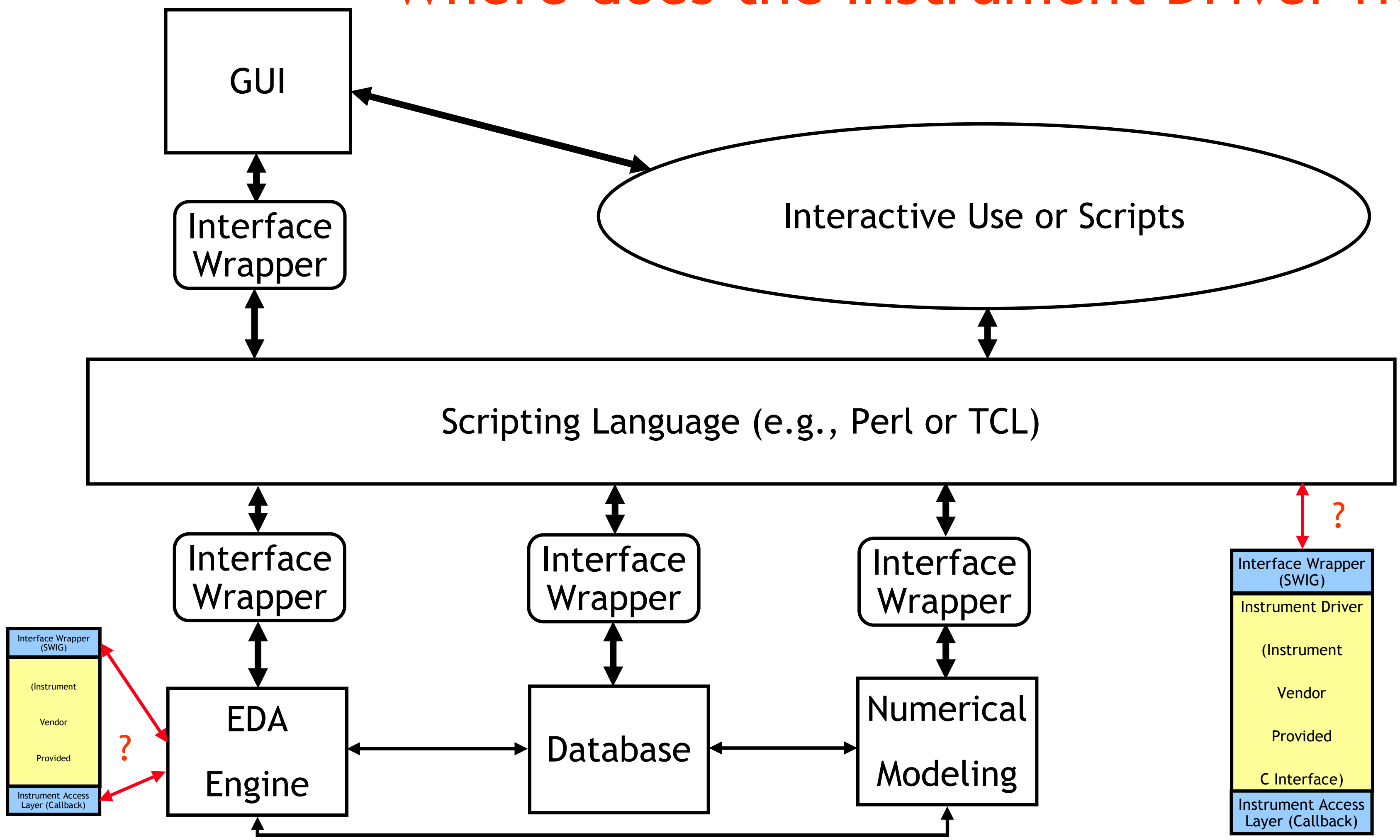


- Instrument control performed by Tool Vendor Application based on aggregate instrument operations over entire chain (concurrency control) through wrapper
- Instrument access performed through Tool Application or direct scan model updates using registered callback handles (not supported by SWIG) for the Instrument Access Layer (e.g., 1687 primitives: GETREG, SETREG, WAIT, …) [Different size registers: Loss of type checking]
- Tool must manage instrument instance data and driver associations

Alcatel·Lucent

Typical EDA Tool Vendor Implementation for Case 4 with proposed “C” Driver

Where does the Instrument Driver fit in?



Alcatel·Lucent

Dynamic Programming Languages (aka, Scripting Languages like Tcl and Python) (From: http://www.tcl.tk/doc/scripting.html)

- Dynamic languages are typically interpreted, highly introspective, and emphasize integration and extension to add new capabilities.
- Scripting languages are intended primarily for plugging together components.
- Scripting languages do their error checking at the last possible moment (execution of that statement).
- Compiled byte code is still an interpreted language where most compilers do not perform semantic validation until run-time via the expression validator.
- Dynamic Programming Language compilers are unable to validate information residing in extension modules written in a different language.
  - If instrument Instr3 is not represented in a C extension, that error will not be reported until the scripted statement using Instr3 is executed, leaving the circuit in a half modified state.
- Dynamic Programming Languages require extensive error handling code that is generally not written by most users.

➢ Script generators can provide continuity of model access across extensions.

Alcatel·Lucent

Utopia: Support Case 4 and Case 5

- NSDL provisions the use of explicit NSDL functional descriptions (with VHDL control flow) and/or delegated native language functional implementations
  - o Michele's logic analyzer and MBIST examples for functional description use
  - o Michele's parallel interface example for delegated native language use
- Allows for compositions of simple instruments to operate as a single complex instrument with a single high level functional interface description leveraging subordinate instrument features/functions (Coordinated hierarchical control in the problem domain) [Something not possible with C proposal]
  - This feature allows for fast integration of instrument blocks to create a more complex coordinated instrument using basic building block designs
- Allows for proprietary design integration using the same library mechanisms of VHDL or the delegated native language functional implementation

➢It is possible to achieve the best of both worlds in a single unified solution!

Alcatel·Lucent

The NSDL Process Flow

LabView Model

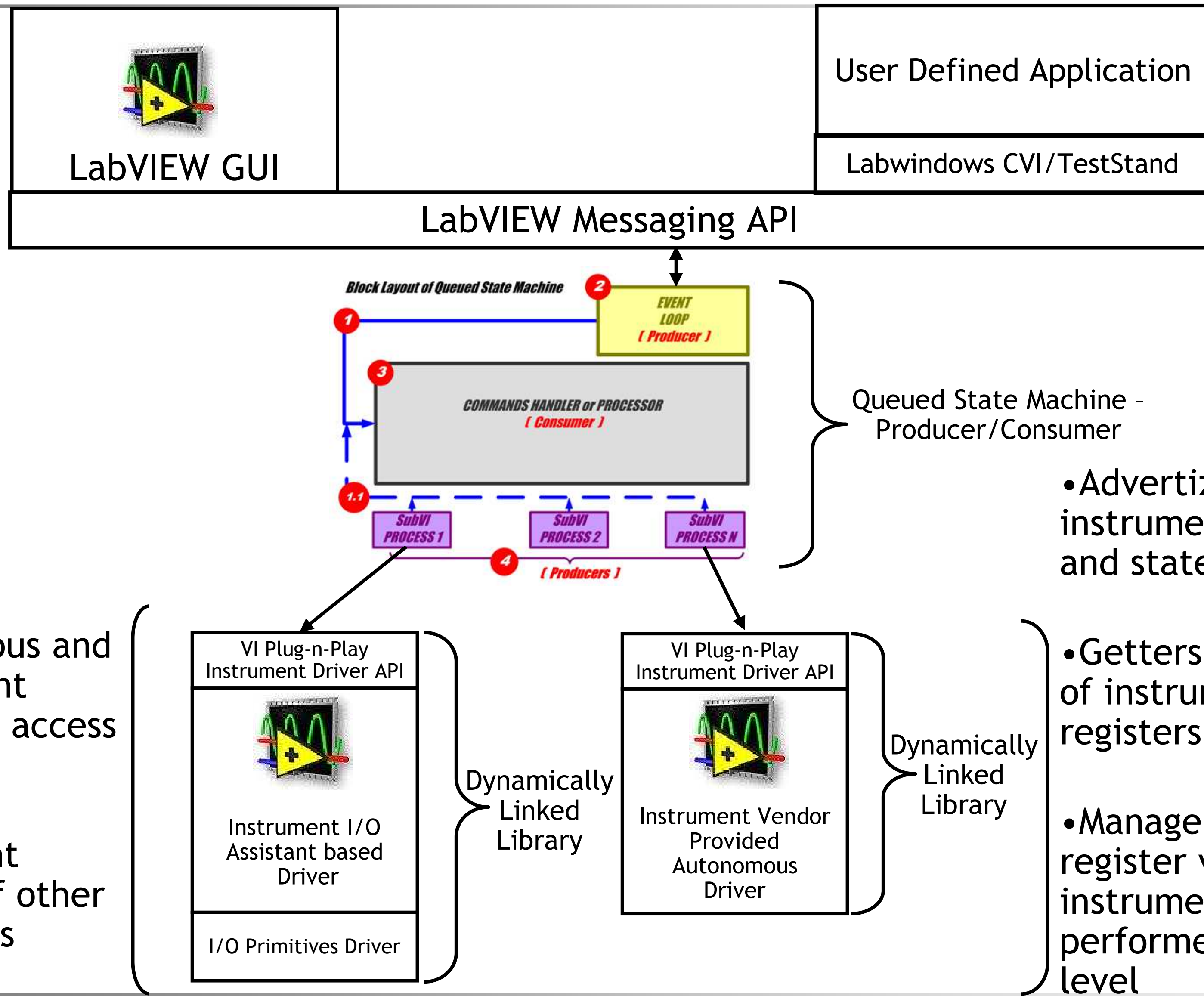
Bradford G. Van Treuren and Michele Portolan

November 2008

LabVIEW Basics

- Instruments are represented as:
  - ✓ a set of registers
  - ✓ a set of states
- Instruments post events to a Queued State Machine as instrument states are changed
- Instrument states may change as a result of a command event from the Queued State Machine (e.g., a change request in a register value)
- Instruments may also post events due to internal change events within the driver software (e.g., interrupt handler events)
- All access to instrument information done using proprietary LabVIEW messaging API

Virtual Instrument Driver



- Advertizes instrument registers and states

- Autonomous and independent instrument access protocols
- Instrument unaware of other instruments

- Getters and Setters of instrument registers and states
- Management of register values and instrument state performed at higher level

Alcatel·Lucent

Contrast / Comparison

LabVIEW

- Requires independent and autonomous access mechanism/protocol to instruments
- Does not require modeling access path because all instruments are represented as registers and states
- Dependent on NI Queued State Machine and messaging API

1687

- Requires shared access mechanism/protocol with dependence on access state of other instruments
- Requires modeling of access path due to dependence on access state of other instruments
- Requires integration with tool board models because of dependence on board chain access path

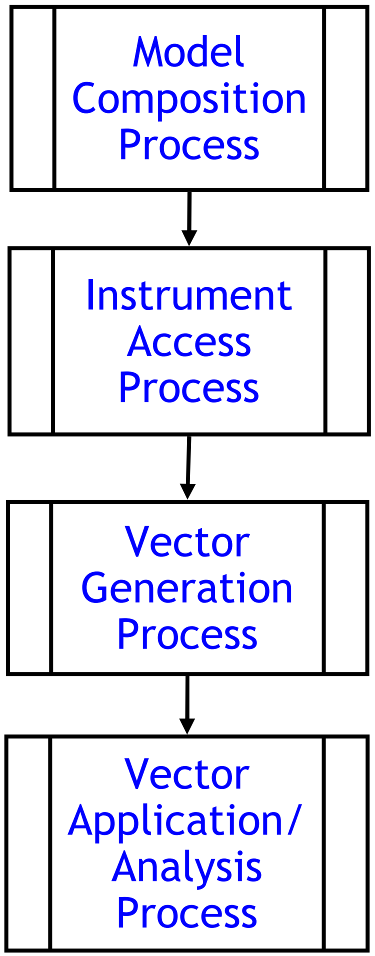
The NSDL Process Flow

Generalized Batch Process Flow

Bradford G. Van Treuren and Michele Portolan

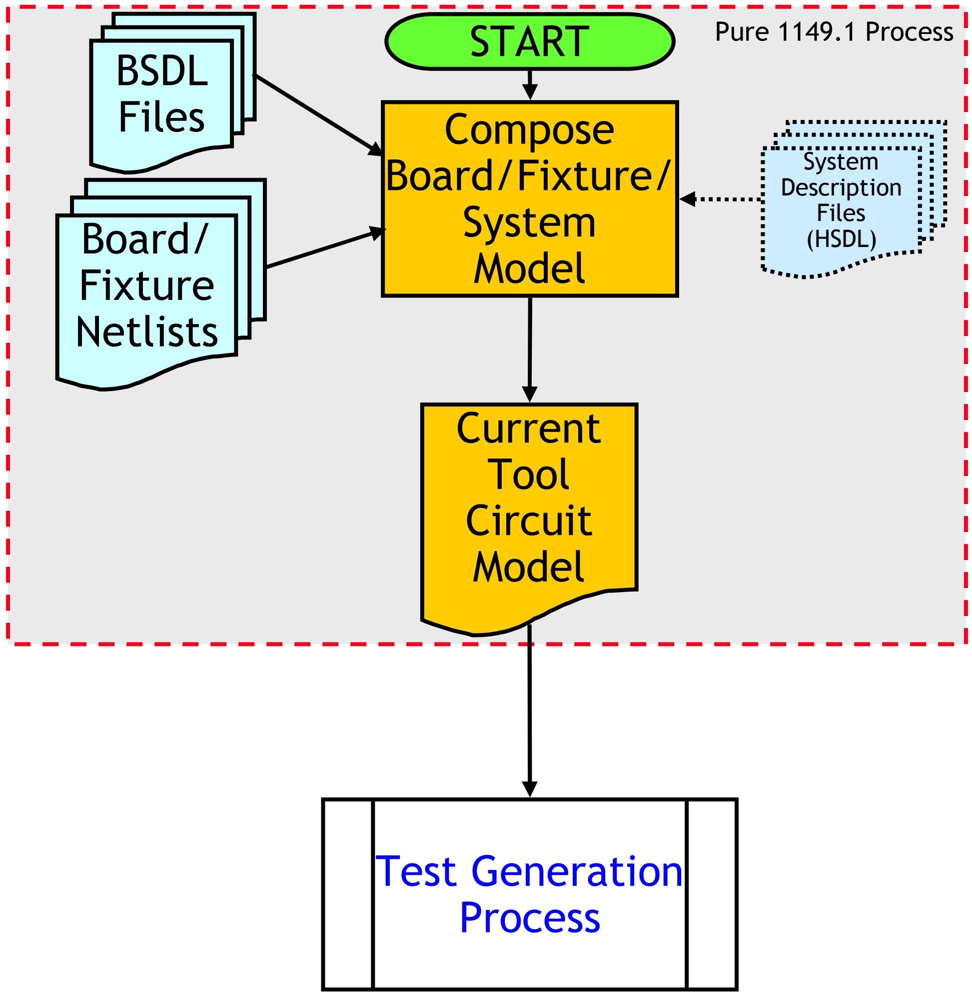
November 2008

Tool Integration Process



Typical Model Composition Process

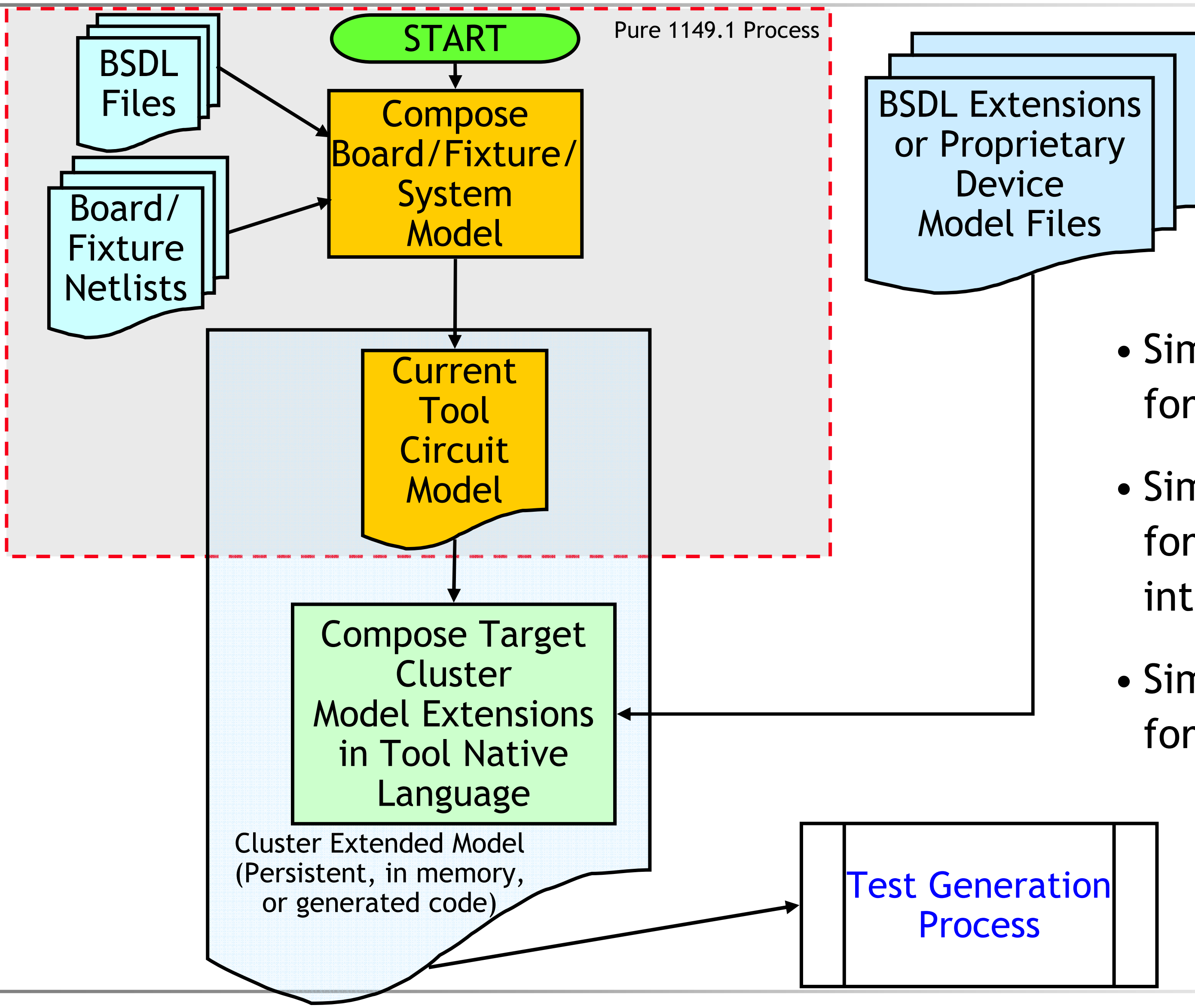
Board/System Level Perspective : current 1149.1 tools



Alcatel·Lucent

Typical Model Composition Process

Non-BScan Cluster Level Perspective : current 1149.1 tools proprietary implementation

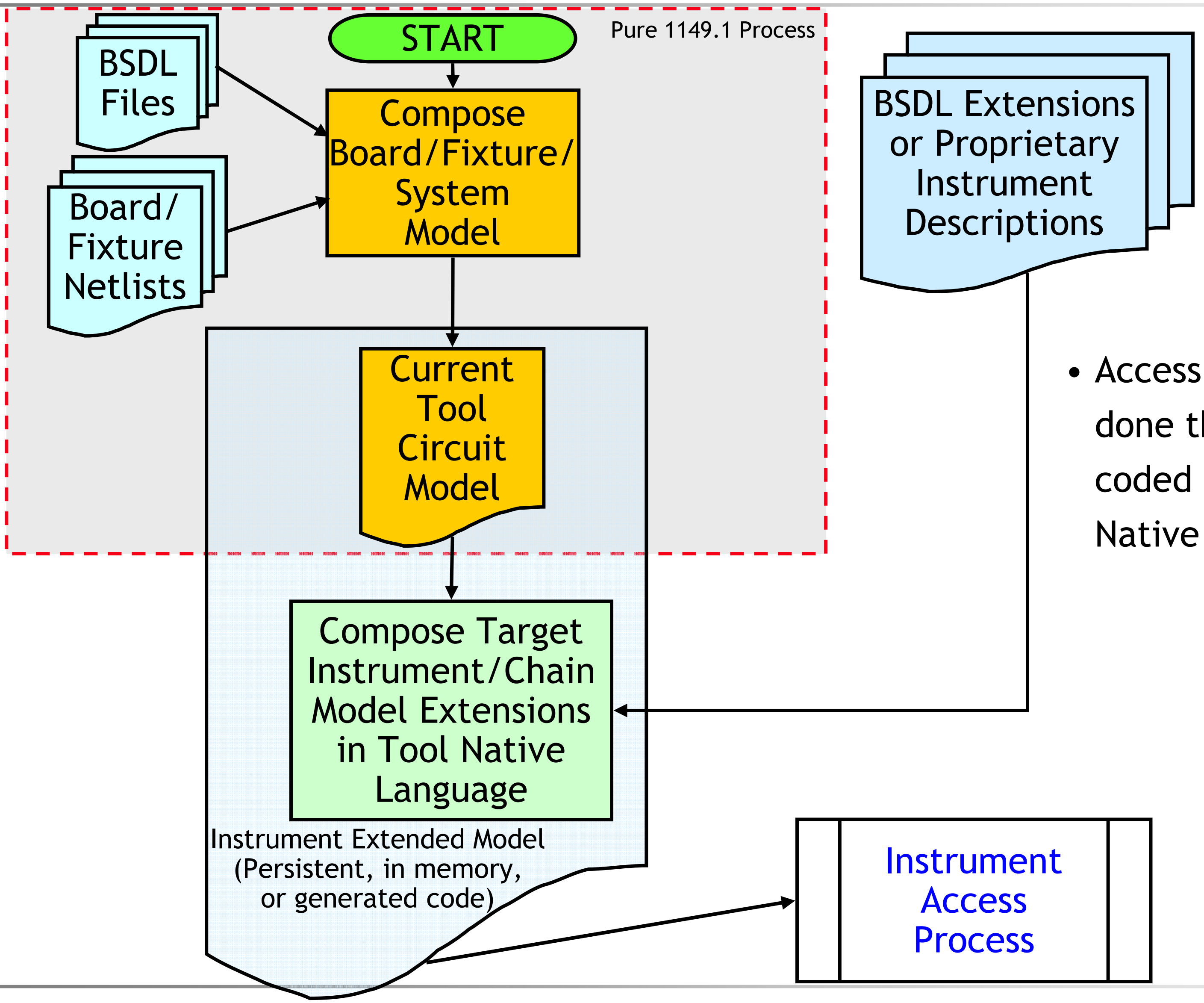


- Similar process for IEEE 1149.6
- Similar process for memory interconnect test
- Similar process for IEEE 1532

Alcatel·Lucent

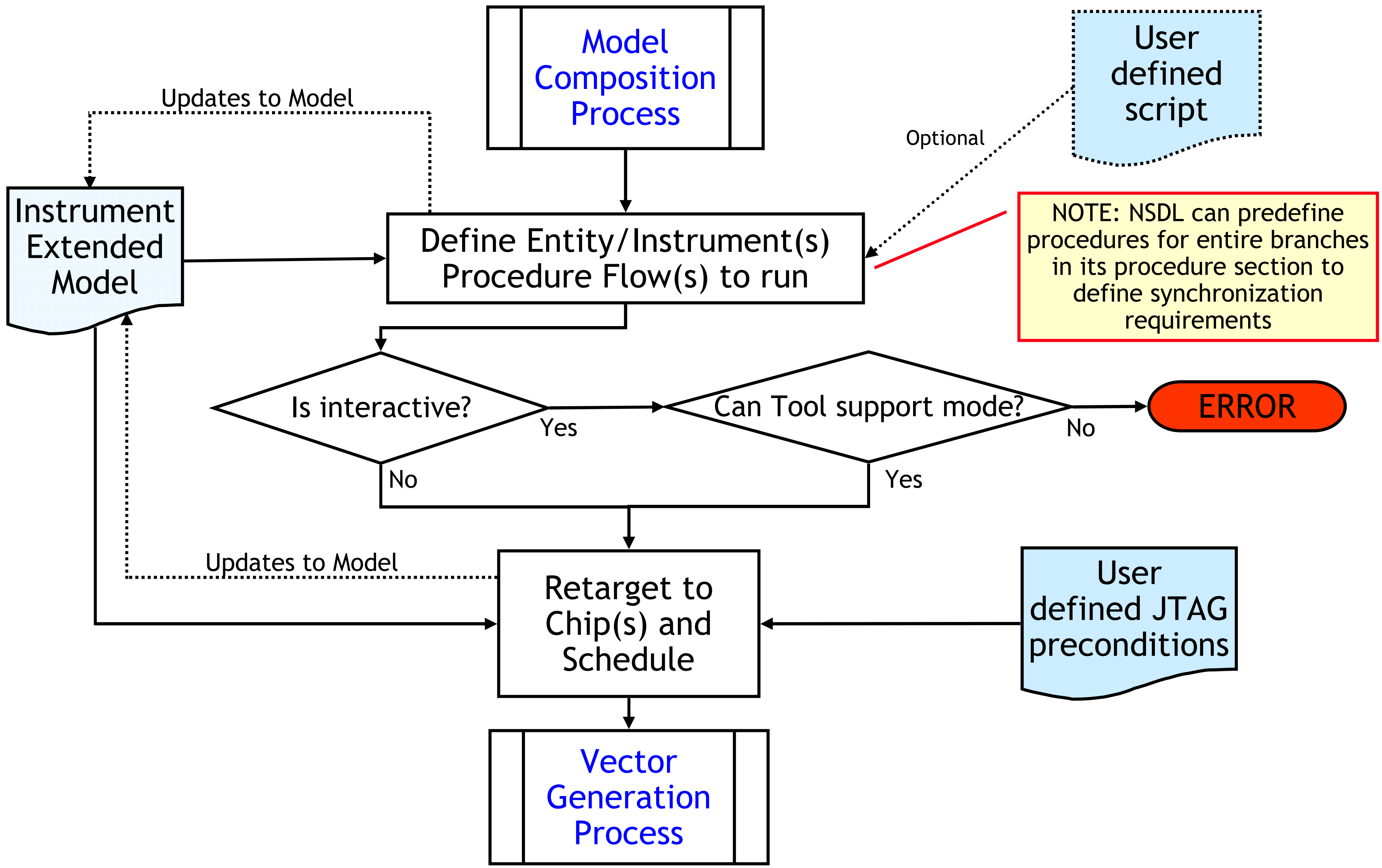
Typical Model Composition Process

Instrument Level Perspective : current 1149.1 tools proprietary implementation

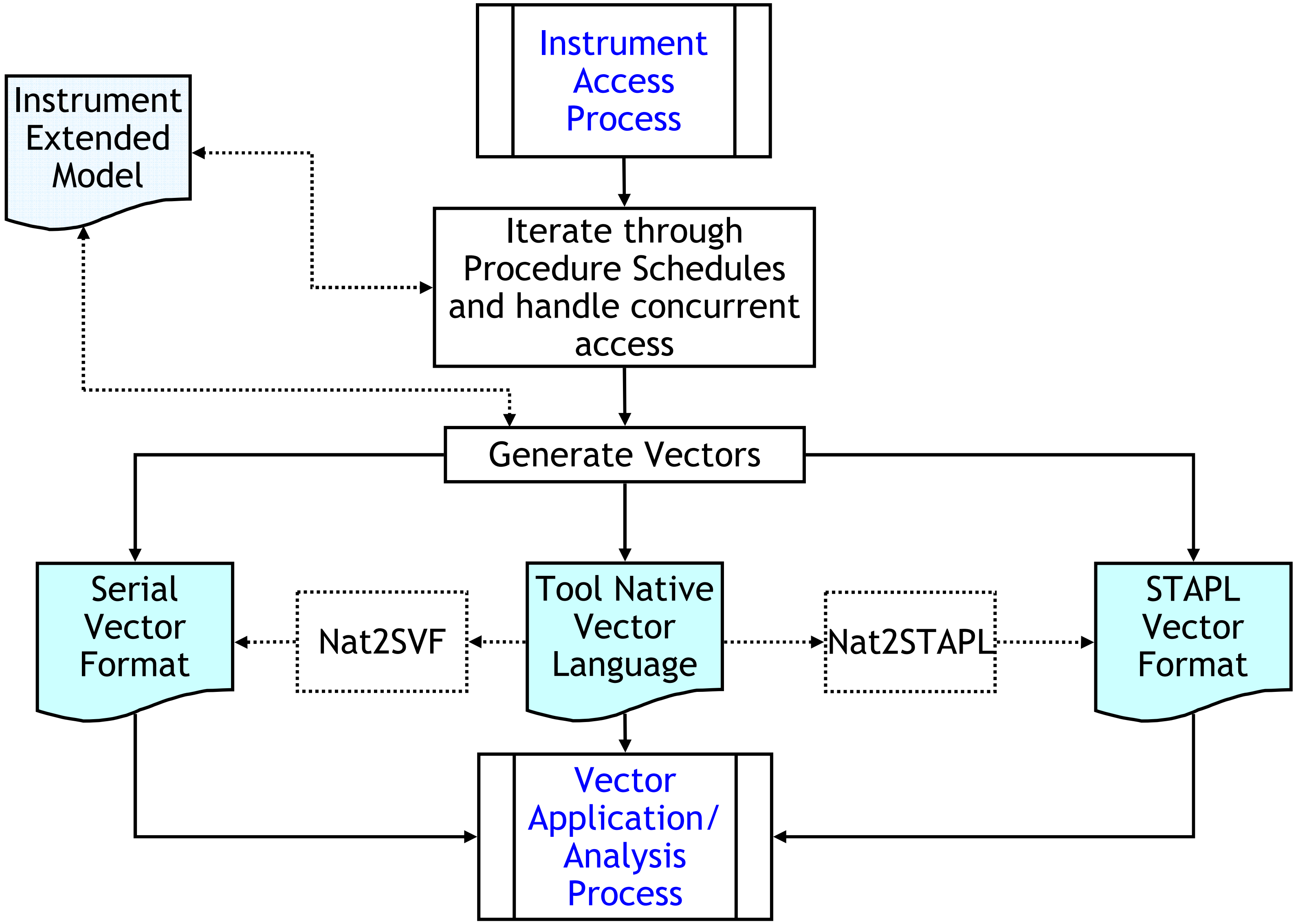


- Access to instruments done through hand coded programs in Tool Native Language

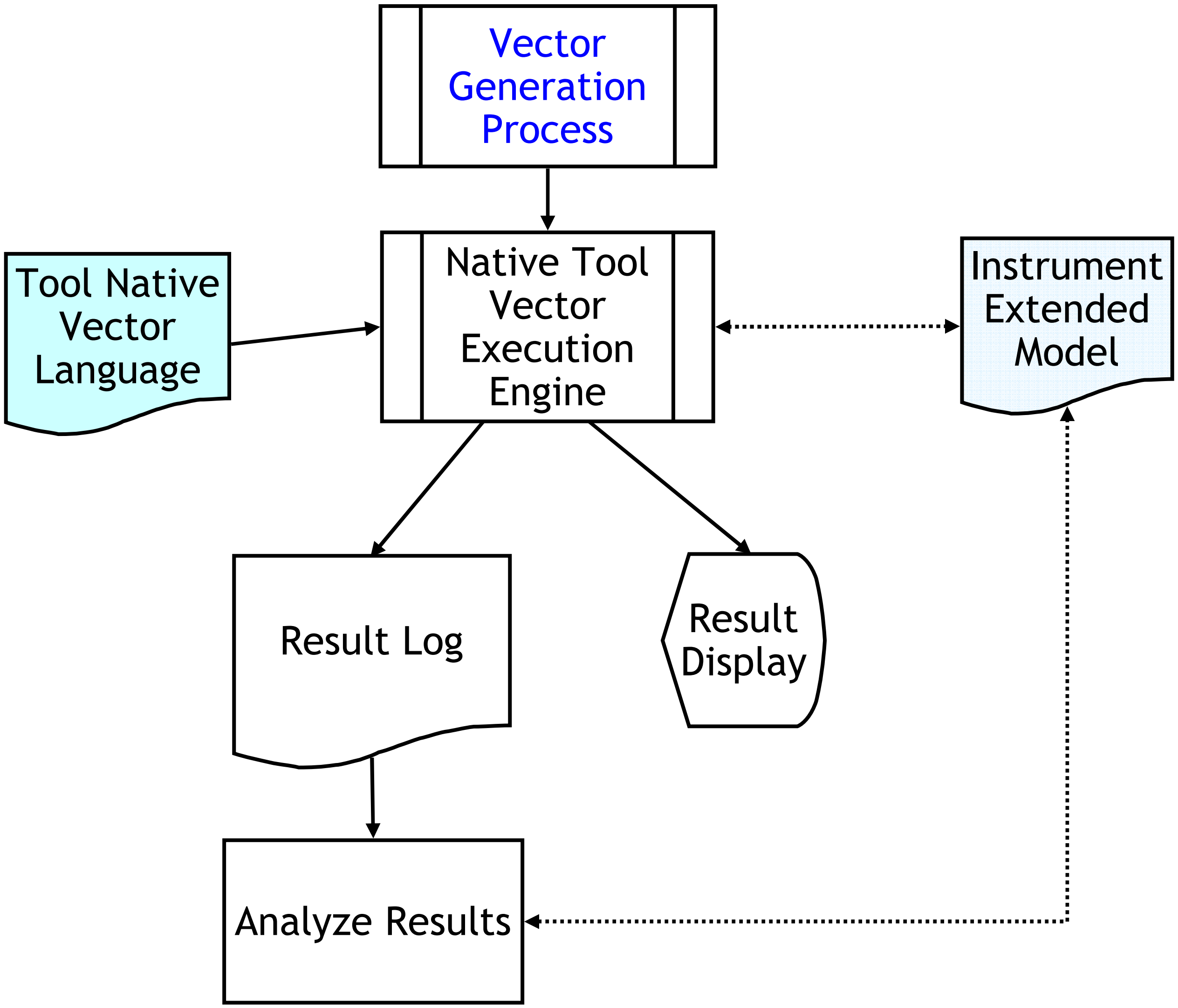
Instrument Access Process



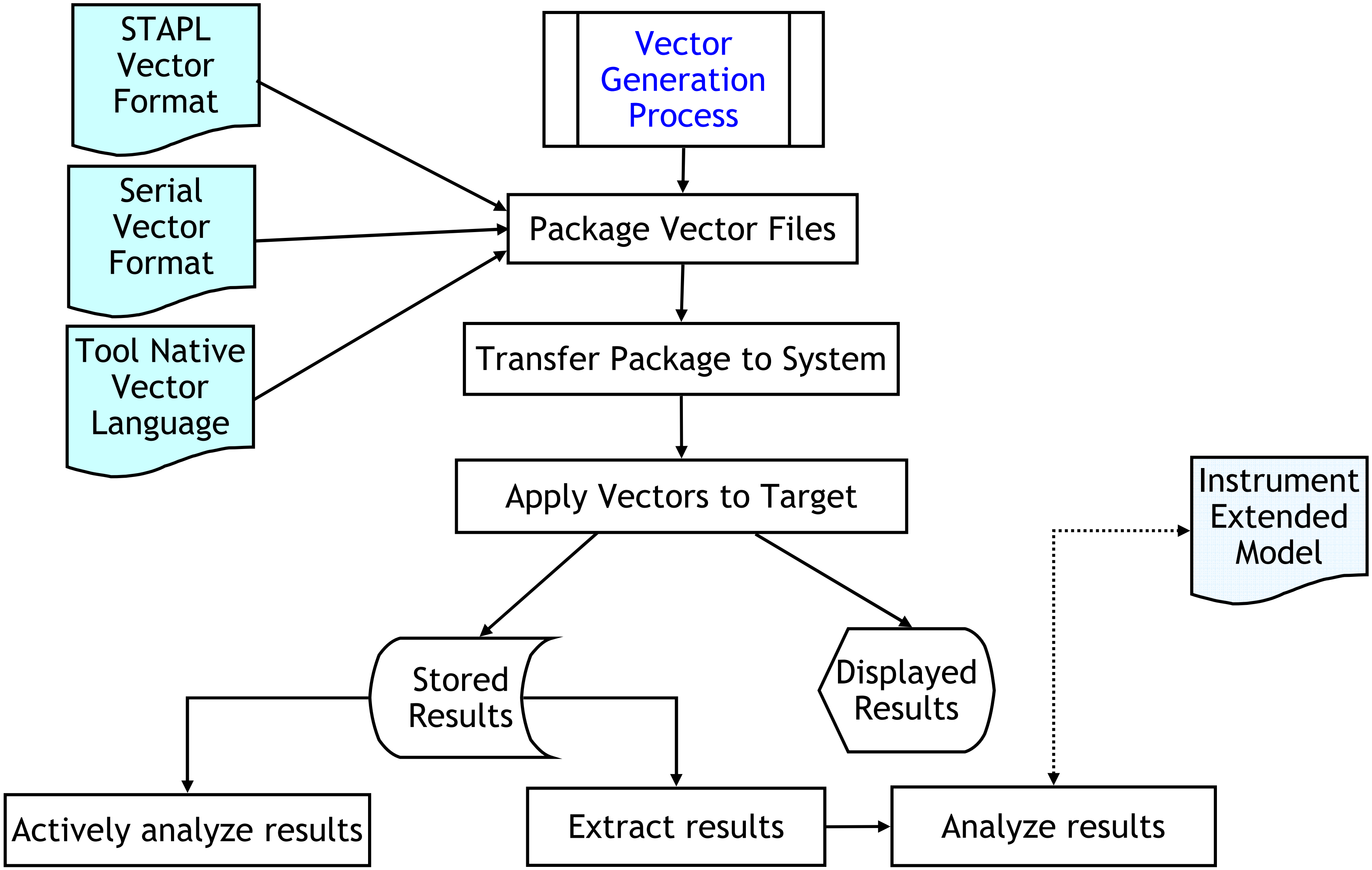
Vector Generation Process



Native Vector Application/Analysis Process



Embedded System Vector Application/Analysis Process



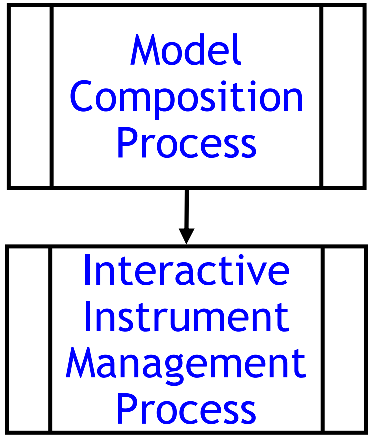
The NSDL Process Flow

Generalized Interactive Process Flow

Bradford G. Van Treuren and Michele Portolan

November 2008

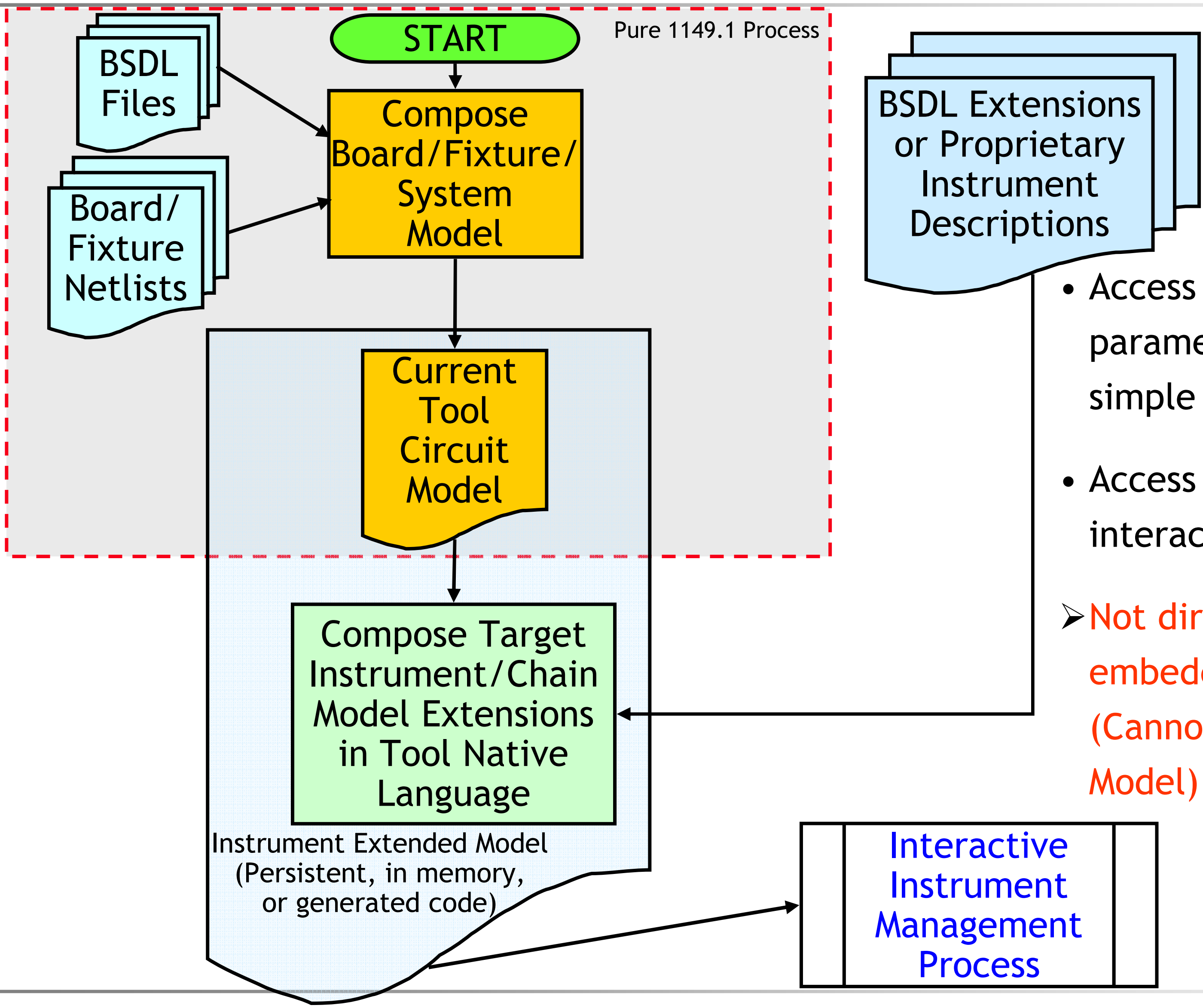
Interactive Tool Integration Process



Alcatel·Lucent

Typical Model Composition Process

Instrument Level Perspective : current 1149.1 tools proprietary implementation

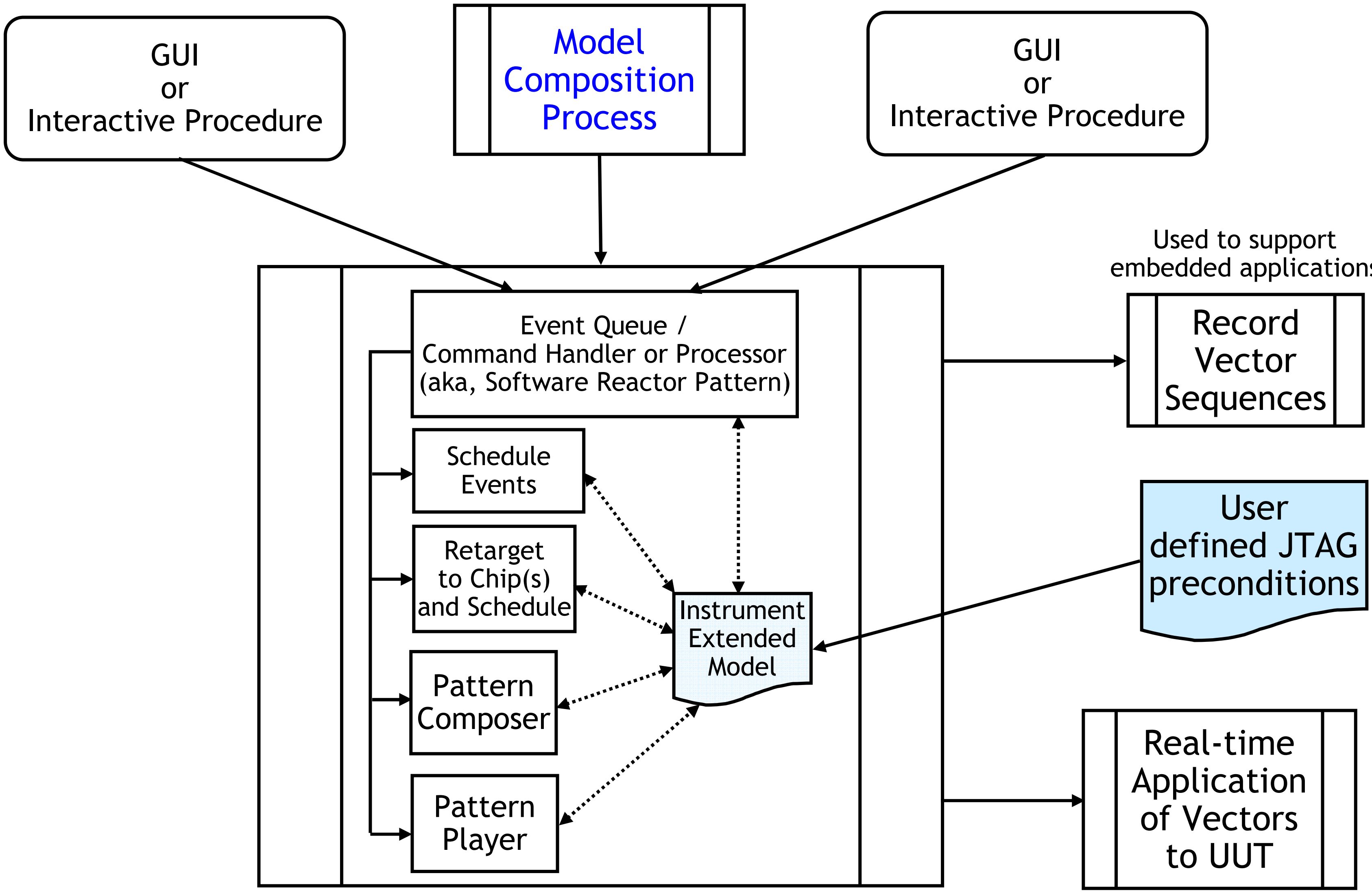


- Access to instruments via parameterized GUI for simple instruments
- Access to instruments via interactive program files

➢ Not directly applicable to embedded application (Cannot replicate Circuit Model)

Alcatel·Lucent

Interactive Instrument Management Process



Alcatel·Lucent

Alcatel·Lucent

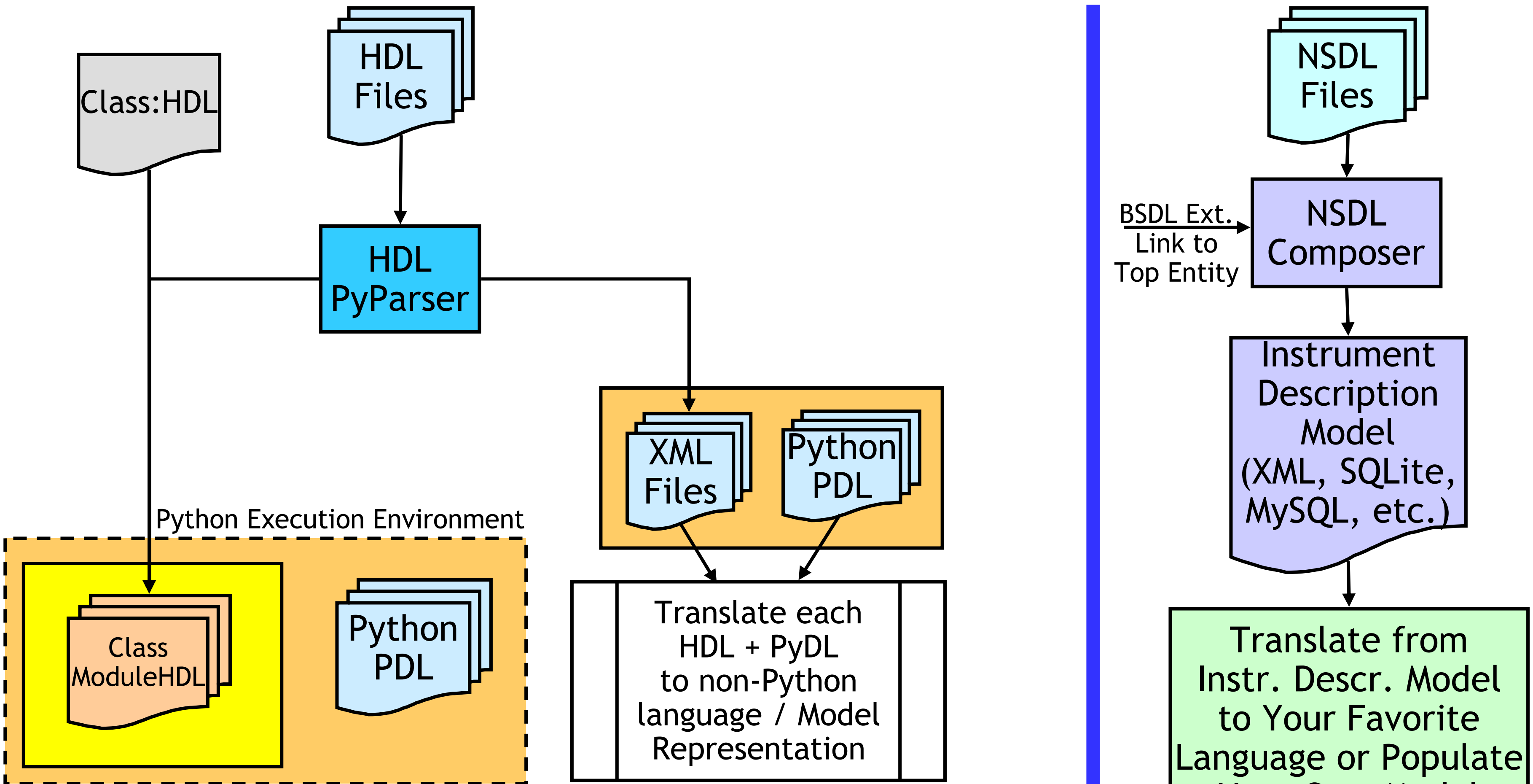
The NSDL Process Flow

NSDL Instrument Description Model

Bradford G. Van Treuren and Michele Portolan

November 2008

HDL/PyDL ⇔ NSDL Comparison



- Modeled in Python executable directly
- Void of Tool TAP Controller Knowledge

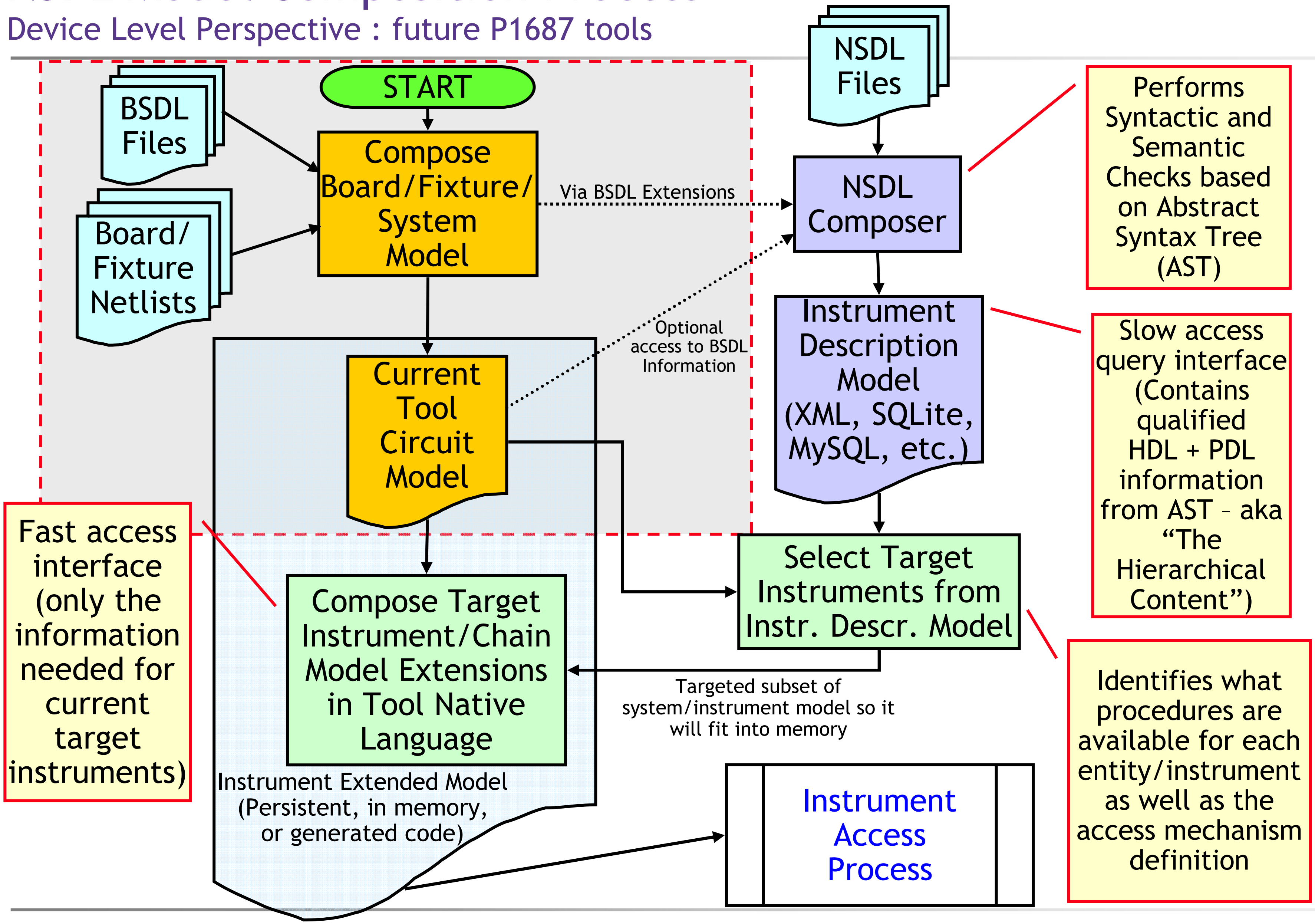
- Multiple Association Representation Model

- Unified Representation Model

Alcatel·Lucent

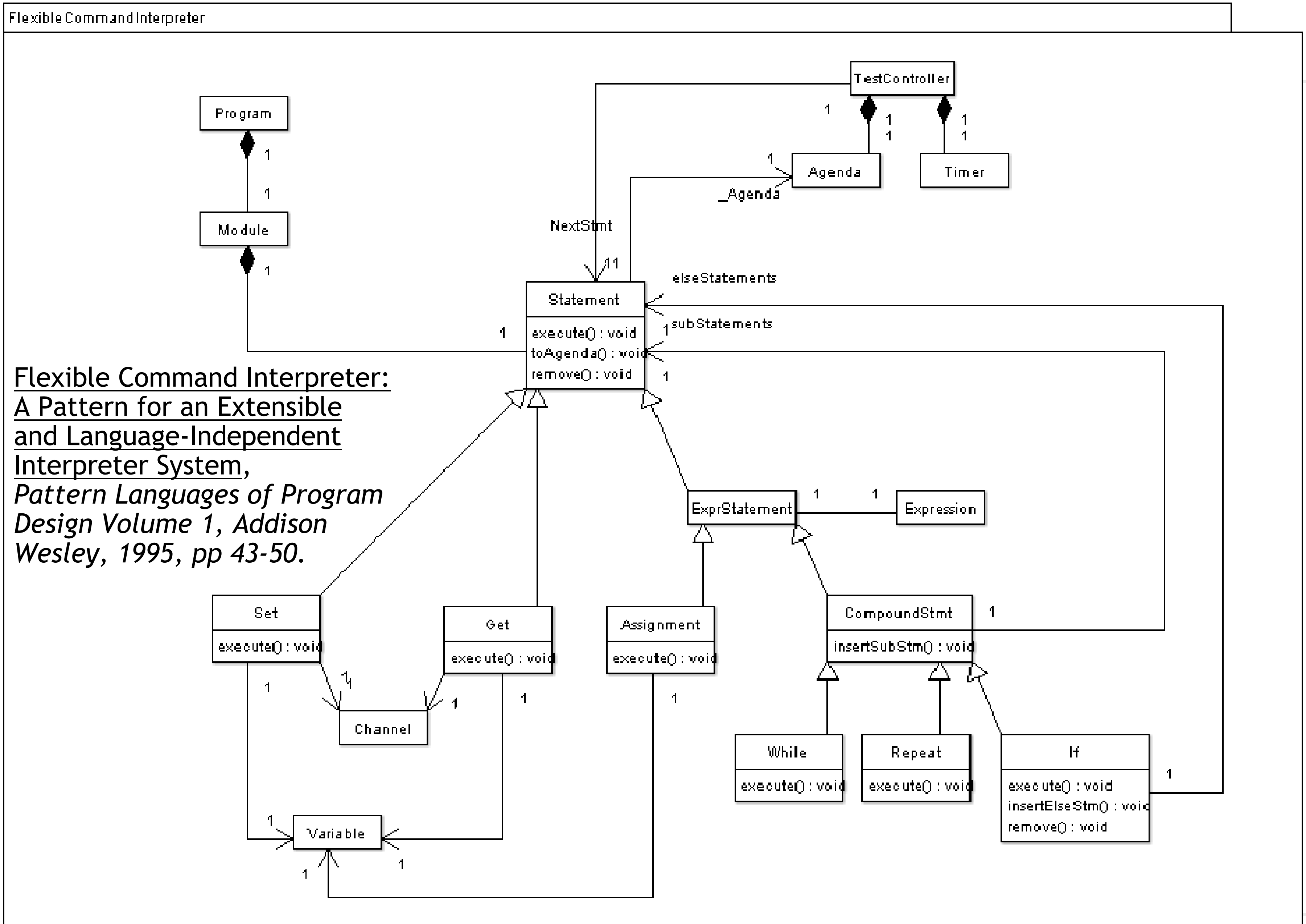
NSDL Model Composition Process

Device Level Perspective : future P1687 tools



Alcatel·Lucent

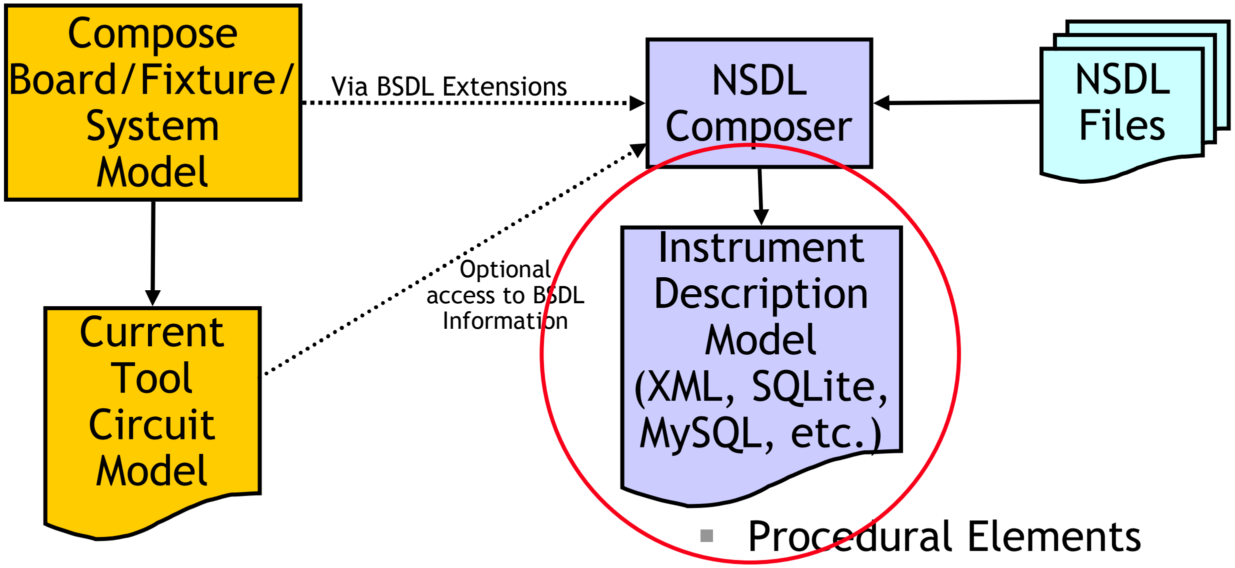
Software Design Pattern: Flexible Command Interpreter for Test Languages



<u>Flexible Command Interpreter:
A Pattern for an Extensible
and Language-Independent
Interpreter System</u>,
Pattern Languages of Program Design Volume 1, Addison Wesley, 1995, pp 43-50.

Alcatel·Lucent

NSDL Instrument Description Model



- Structural Elements
  - Ports
  - Registers
  - Attributes
  - Cell Types
  - Instances
  - Etc.

- Procedural Elements
  - Procedures
  - Ordered Statements (Queue for Agenda)
    - Set
    - Get
    - Assign
    - While
    - ExpressionStmt
    - Etc.
  - Variables
  - Synchronization dependencies
  - Etc.

Alcatel·Lucent

NSDL Instrument Procedural Description Model Elements

The Translator Implementation

- Abstract syntax tree view
- Semantic inference about usage (e.g., interactive vs. deferred)
- Semantic validation of tree dependency structure (e.g., instances are properly defined)
- Referenced procedural element dependencies are validated
- Statements are represented as ordered XML objects
- Expressions are contained and applied as XML objects in the corresponding ExprStatement subclass representations
- Expressions are ordered based on Abstract syntax tree to ensure correct precedence ordering
- Boolean logic in Expressions maps directly to native language boolean logic
- Language translator implements execute( ) function for each statement type
- Interpreter executes statements “in order” to write out translated file in native language format
- Parametric information wrapped inside containing statement

FCI Example of NSDL to C++ Translation

```
➢data1 := ‘1’;

➢If sel1 and sel2 and not sel3 then
   instr3.select( );
   reg2 := ‘011001001’;
end if;
```

```
➢data1 = 1;

➢If (sel1 & sel2 & !(sel3)) {
        instr3.select( );
        reg2 = 0xc9;
}
```

```
➢as = AssignmentStatement(lhs=data1, rhs=‘1’);
as.toAgenda( );
as.execute( );  // writes out “lhs = rhs;” when called by TestController

➢ifstmt = IfStatement(Expression(And(sel1,And(sel2, Not(sel3)));
ifstmt.InsertSubStmt(CallStatement(instr=instr3, func=select, args=“”));
ifstmt.InsertSubStmt(AssignmentStatement(lhs=reg2, rhs=‘011001001’));
ifstmt.InsertSubStmt(EndifStatement());
ifstmt.toAgenda( );
ifstmt.execute( ); // writes out “if (“, calls Expression.execute( ), writes out “) {“ then
// calls its SubStmts toAgenda( ) methods “in order” to perform their execute( )
```

Parameterization

- Use of generics to parameterize instantiations and procedures
  - Instruments can define generic values (ex: register length, port width)
  - Defined at instantiation time, high code reuse
    - Ex: generic-width WSP block in “system” example declared as:

```
instrument generic_WSP is
  generic (wrapper_select_signal : integer := 2)
  port (….
```

Can be instantiated as

```
my_wsp_4: generic_WSP generic map (wrapper_select_signal => 4) …
```

or

```
my_wsp_16: generic_WSP generic map (wrapper_select_signal => 16) …
```

- Indexed literals to help deal with high number of identical instances
  - Asic.nsdl in “ericsson” example

    mbist_instance_<i> can be mbist_instance_0, mbist_instance_1, etc..
  - Used in conjunction with generate loops: compact code
  - Flexible and parametrical code

Alcatel·Lucent

Unified language (1)

- Same parameters used for both structural and procedures
  - Flexible and self-contained code
  - Same declaration space: easy to check for automated tool
- All information for a module contained in one place:
  - Easy to debug and human readable
  - Clean and effective partitioning for complex projects
- Functional description removes necessity for structural element keywords
  - Procedure/functions define roles of ports and registers
  - No need to specify it in structural description (hdl)
  - No ambiguity on attribute interpretation
  - No restriction on instrument types

Unified language (2)

- NSDL descriptions are self-contained
  - No notion of TAP : only the P1687 network is described
  - Completely independent and portable descriptions
  - 1687.x would completely reuse current description files

- Synchronisation with external interfaces
  - Ports/registers can describe non-scan paths
  - Associated procedures give the scan-based synchronisation primitives
  - Compatibility with any arbitrary interface

- Inter-instrument communications
  - Port values can be used in functions to define synchronisation points
  - Same thing for dependencies

VHDL heritage

- Well defined types and type generation rules
  - Strict typing checks make code robust
  - No ambiguities in procedures: portable between implementations
  - Possibilities of exploiting VHDL hardware-oriented types (std_logic, ulogic, integers, unsigned, etc…)
- Extensively verified and robust syntax
  - Years of use make for unambiguous interpretation
- Hardware-oriented language
  - Terse and unambiguous syntax
  - Natural support of hierarchy and point-to-point connections
  - Hardware flow friendly: architectures and configurations can be adapted to each step from manufacture to field use

Alcatel·Lucent

The NSDL Process Flow

Backup Slides

Bradford G. Van Treuren and Michele Portolan

November 2008

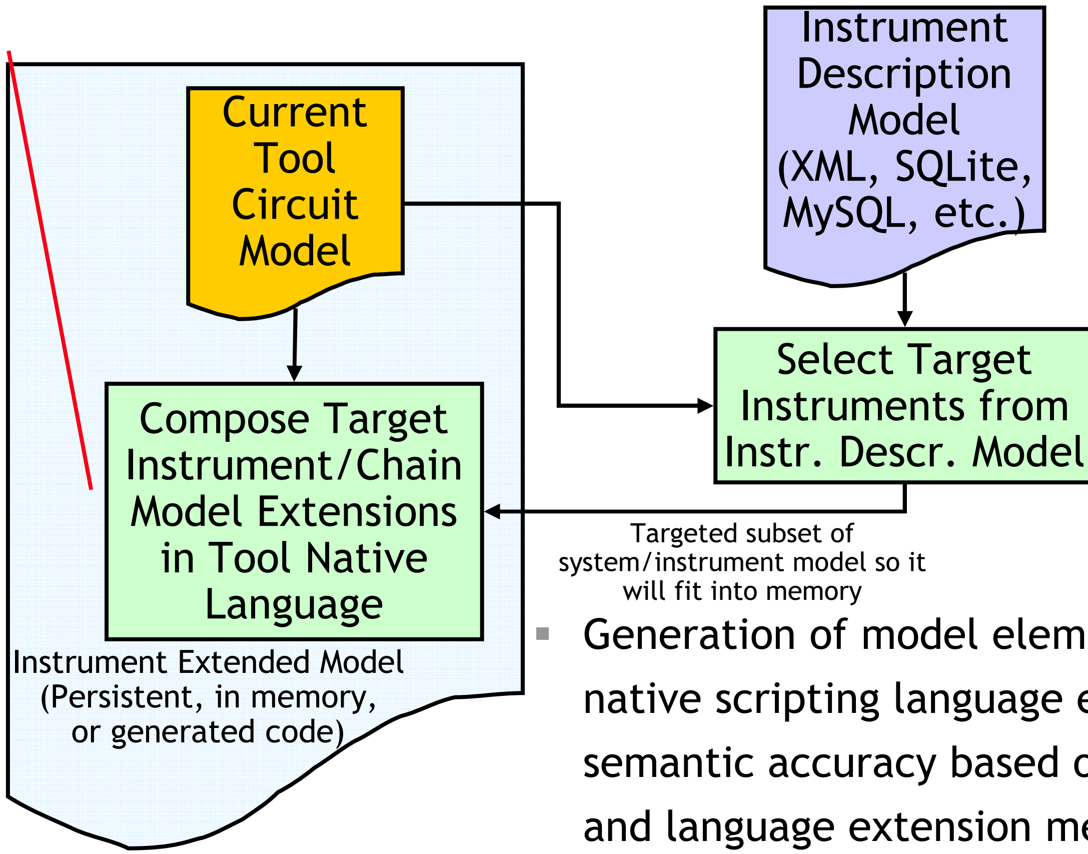
Dynamic Programming Languages (aka, Scripting Languages like Tcl and Python) (From: http://www.tcl.tk/doc/scripting.html)

- Dynamic languages are typically interpreted, highly introspective, and emphasize integration and extension to add new capabilities.
- Scripting languages are intended primarily for plugging together components.
- Scripting languages do their error checking at the last possible moment (execution of that statement).
- Compiled byte code is still an interpreted language where most compilers do not perform semantic validation until run-time via the expression validator.
- Dynamic Programming Language compilers are unable to validate information residing in extension modules written in a different language.
  - If instrument Instr3 is not represented in a C extension, that error will not be reported until the scripted statement using Instr3 is executed, leaving the circuit in a half modified state.
- Dynamic Programming Languages require extensive error handling code that is generally not written by most users.

➢ Script generators can provide continuity of model access across extensions.

Relation of Instrument Description Model to Dynamic Programming Languages

- Tool vendor creates their own circuit model used by their tool
- Instrument provider unable to define each tool vendor’s model elements unless model access is standardized
- Entity structure is possible to represent generically
- Procedural interface must still be defined by the instrument provider to obtain proper language extension access
- Instrument access inside procedures is based on Tool Model access definition

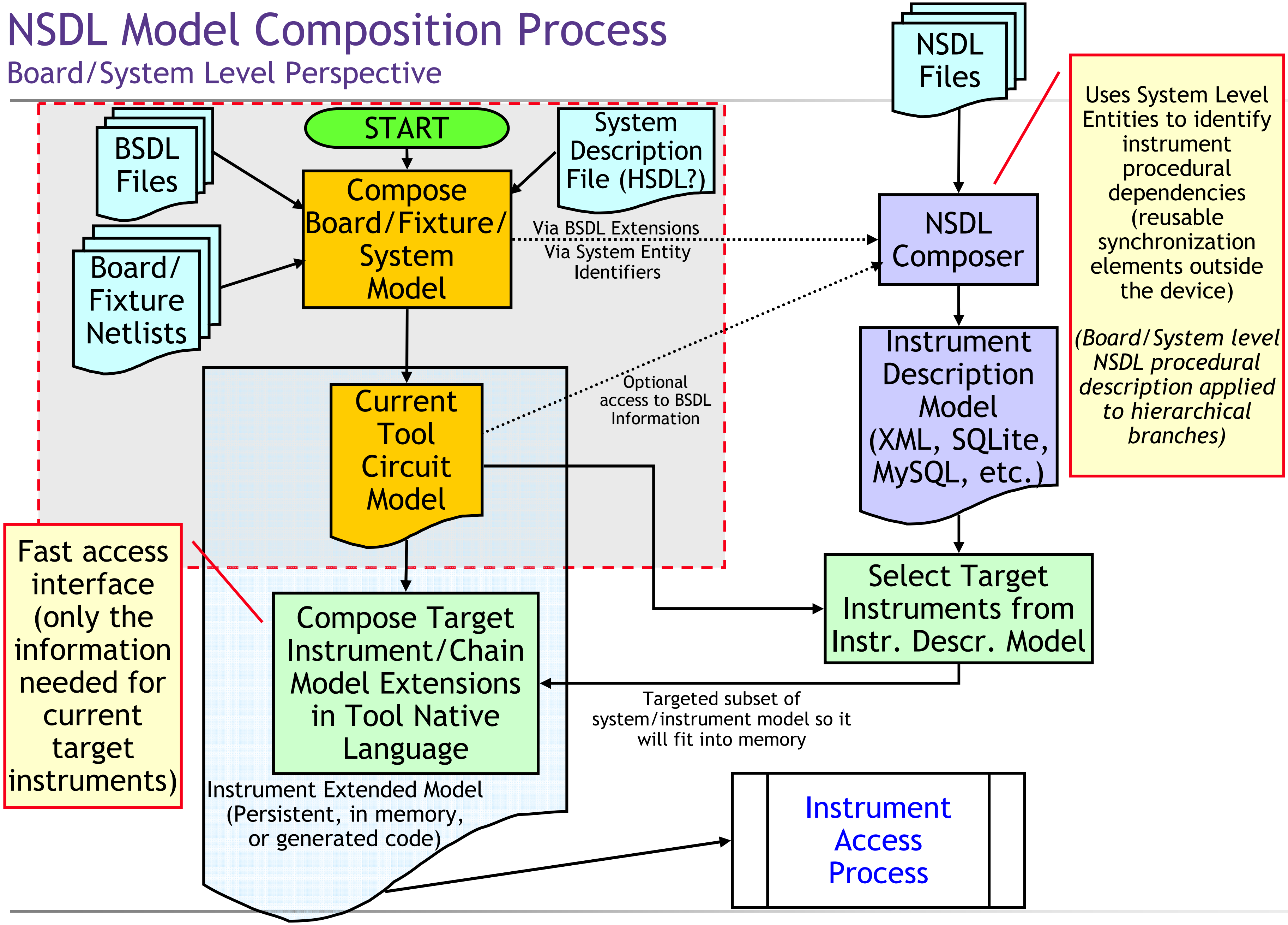


- May compose native circuit model in tool’s software language based on structure and procedural elements defined in the NSDL IDM.
- Generation of model elements in native scripting language ensures semantic accuracy based on NSDL IDM and language extension mechanisms.
- May generate scripting language extensions in C/C++ dynamically based on common API for model information access.

Alcatel·Lucent

NSDL Model Composition Process

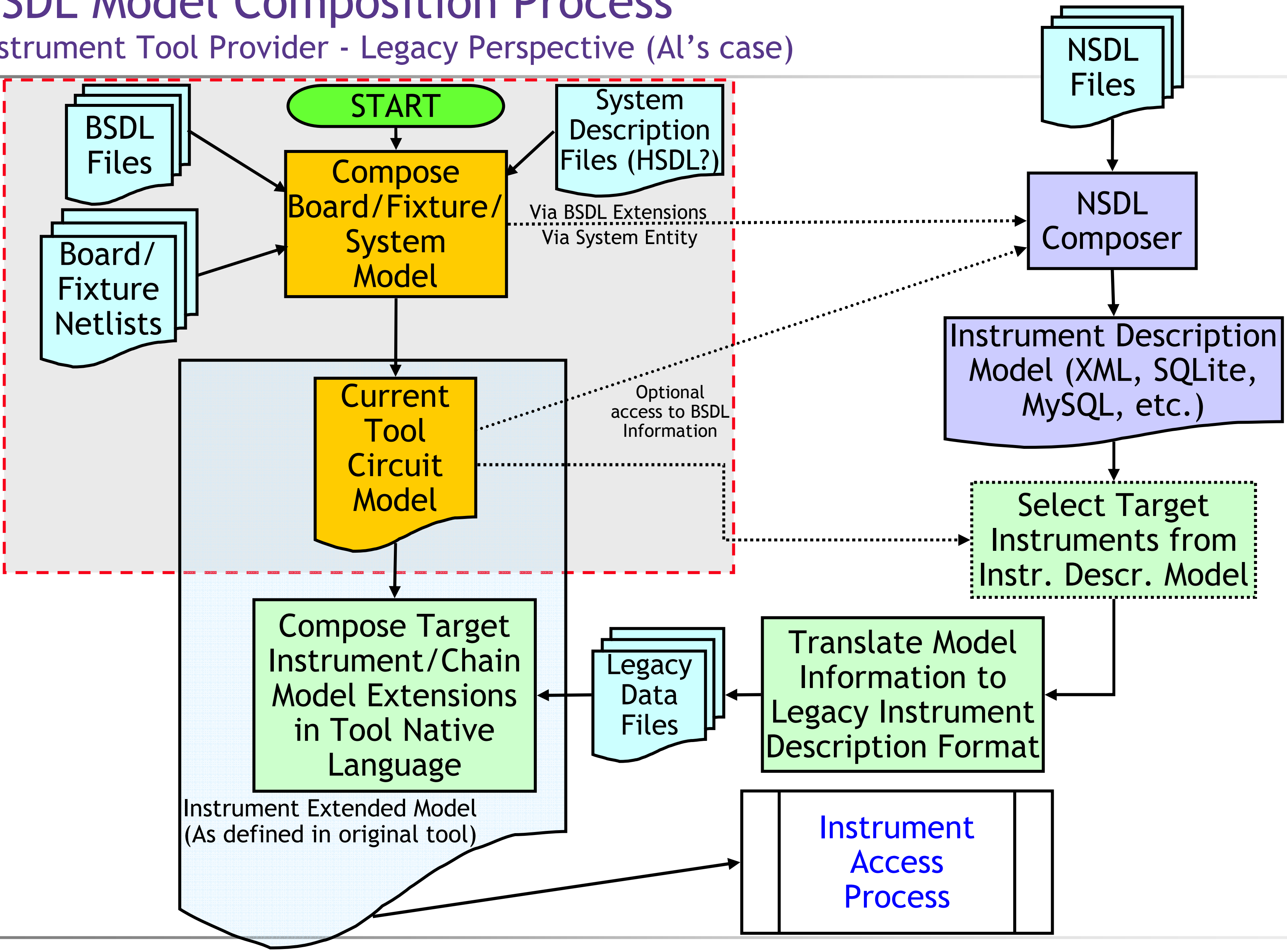
Board/System Level Perspective



Alcatel·Lucent

NSDL Model Composition Process

Instrument Tool Provider - Legacy Perspective (Al's case)



Alcatel·Lucent