Architecture description languages for programmable embedded systems

P. Mishra and N. Dutt

Abstract: Embedded systems present a tremendous opportunity to customise designs by exploiting the application behaviour. Shrinking time-to-market, coupled with short product lifetimes, create a critical need for rapid exploration and evaluation of candidate architectures. Architecture description languages (ADL) enable exploration of programmable architectures for a given set of application programs under various design constraints such as area, power and performance. The ADL is used to specify programmable embedded systems, including processor, coprocessor and memory architectures. The ADL specification is used to generate a variety of software tools and models facilitating exploration and validation of candidate architectures. The paper surveys the existing ADLs in terms of (*a*) the inherent features of the languages and (*b*) the methodologies they support to enable simulation, compilation, synthesis, test generation and validation of programmable embedded systems. It concludes with a discussion of the relative merits and demerits of the existing ADLs and expected features of future ADLs.

1 Introduction

Embedded systems are everywhere – they run the computing devices hidden inside a vast array of everyday products and appliances, such as cell phones, toys, handheld PDAs, cameras and microwave ovens. Cars are full of them, as are airplanes, satellites and advanced military and medical equipments. As applications grow increasingly complex, so do the complexities of the embedded computing devices. Figure 1 shows an example embedded system, consisting of programmable components including a processor core, coprocessors and memory subsystem. The programmable components are used to execute the application programs. Depending on the application domain, the embedded system can have application specific hardwares, interfaces, controllers, and peripherals. In this paper, we refer to the programmable components, consisting of a processor core, coprocessors and memory subsystem, as *programmable embedded systems*. We also refer to them as *programmable architectures*.

As embedded systems become ubiquitous, there is an urgent need to facilitate rapid design space exploration (DSE) of programmable architectures. This need for rapid DSE becomes even more critical given the dual pressures of shrinking time-to-market and ever-shrinking product lifetimes. Architecture description languages (ADL) are used to perform early exploration, synthesis, test generation, and validation of processor-based designs as shown in Fig. 2. ADLs are used to specify programmable architectures. The specification can be used for generation of a software toolkit including the compiler, assembler, simulator and debugger. The application programs are compiled and simulated, and the feedback is used to modify the ADL specification with the goal of finding the best possible architecture for the given set of applications. The ADL specification can also be used for generating hardware prototypes under design constraints such as area, power and clock speed. Several researches have shown the usefulness of ADL-driven generation of functional test programs and test interfaces. The specification can also be used to generate device drivers for real-time operating systems.

Previously, researchers have surveyed architecture description languages for retargetable compilation [1] and systems-on-chip design [2]. Qin and Malik [1] surveyed the existing ADLs and compared the ADLs to highlight their relative strengths and weaknesses in the context of retargetable compilation. Tomiyama *et al.* [2] classified existing ADLs into four categories based on their main objectives: synthesis, compiler generation, simulator generation and validation. This paper presents a comprehensive survey of existing ADLs and the accompanying methodologies for programmable embedded systems design.

2 ADLs and other languages

The term 'architecture description language' (ADL) has been used in the context of designing both software and hardware architectures. Software ADLs are used for representing and analysing software architectures [3, 4]. They capture the behavioural specifications of the components and their interactions that comprise the software architecture. However, hardware ADLs capture the structure (hardware components and their connectivity), and the behaviour (instruction set) of processor architectures. This paper surveys hardware ADLs.

The concept of using machine description languages for specification of architectures has been around for a long time. Early ADLs such as ISPS [5] were used for simulation, evaluation and synthesis of computers and other digital systems. In this paper, we survey contemporary ADLs.

IEE Proceedings online no. 20045071

doi: 10.1049/ip-cdt:20045071

Paper first received 5th July 2004 and in revised form 27th January 2005

P. Mishra is with the Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32611, USA

N. Dutt is with the Center for Embedded Computer Systems, University of California, Irvine, CA 92697, USA

E-mail: prabhat@cise.ufl.edu

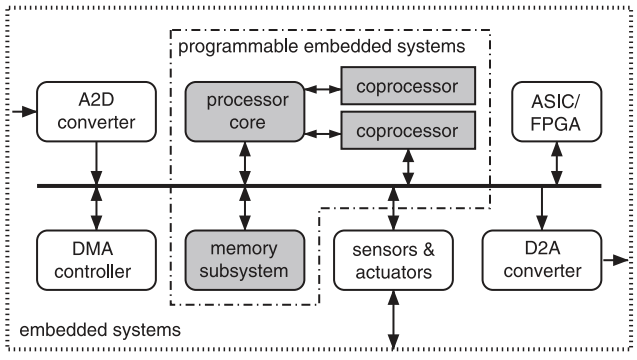

Fig. 1 *Example embedded system*

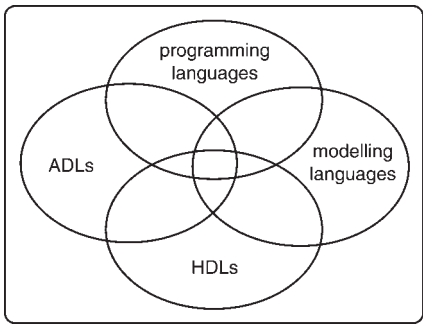

Fig. 3 *ADLs and non-ADLs*

How do ADLs differ from programming languages, hardware description languages, modelling languages, and the like? In this Section, we attempt to answer this question. However, it is not always possible to answer the following question: Given a language for describing an architecture, what are the criteria for deciding whether it is an ADL or not?

In principle, ADLs differ from programming languages because the latter bind all architectural abstractions to specific point solutions whereas ADLs intentionally suppress or vary such binding. In practice, architecture is embodied and recoverable from code by reverse engineering methods. For example, it might be possible to analyse a piece of code written in **C** and figure out whether it corresponds to *Fetch* unit or not. Many languages provide architecture level views of the system. For example, C++ offers the ability to describe the structure of a processor by instantiating objects for the components of the architecture. However, C++ offers little or no architecture-level analytical capabilities. Therefore, it is difficult to describe architecture at a level of abstraction suitable for early analysis and exploration. More importantly, traditional programming languages are not the natural choice for describing architectures due to their inability to capture hardware features such as parallelism and synchronisation.

ADLs differ from modelling languages (such as UML) because the latter are more concerned with the behaviours of the whole rather than the parts, whereas ADLs concentrate on representation of components. In practice, many modelling languages allow the representation of co-operating components and can represent architectures reasonably well. However, the lack of an abstraction would make it harder to describe the instruction set of the architecture.

Traditional hardware description languages (HDL), such as VHDL and Verilog, do not have sufficient abstraction to describe architectures and explore them at the system level. It is possible to perform reverse-engineering to extract the structure of the architecture from the HDL description. However, it is hard to extract the instruction set behaviour of the architecture. In practice, some variants of HDLs work reasonably well as ADLs for specific classes of programmable architectures.

There is no clear line between ADLs and non-ADLs. In principle, programming languages, modelling languages and hardware description languages have aspects in common with ADLs, as shown in Fig. 3. Languages can, however, be discriminated from one another according to how much architectural information they can capture and analyse. Languages that were born as ADLs show a clear advantage in this area over languages built for some other purpose and later co-opted to represent architectures. We revisit this issue in Section 5 in light of the survey results.

3 ADL survey

Figure 4 shows the classification of architecture description languages (ADLs) based on two aspects: *content* and *objective*. The content-oriented classification is based on the nature of the information an ADL can capture, whereas the objective-oriented classification is based on the purpose of an ADL. Contemporary ADLs can be classified into six categories based on the objective: simulation-oriented,

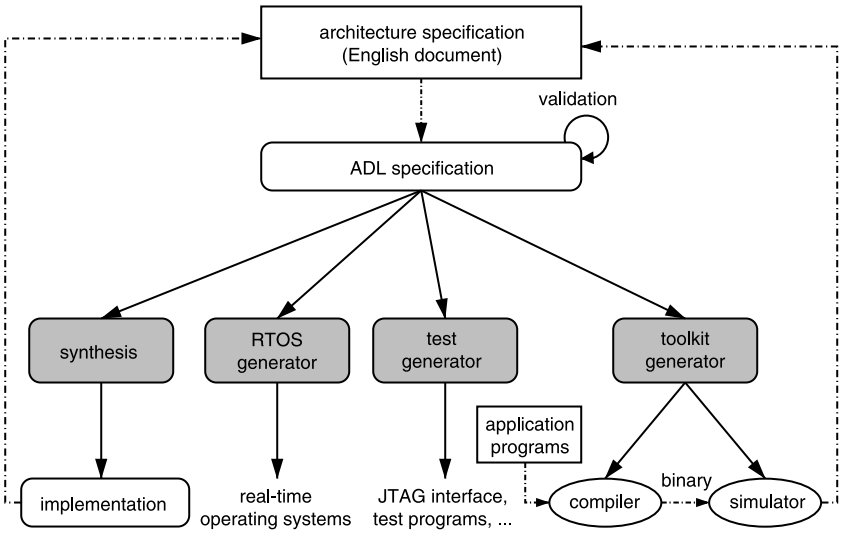

Fig. 2 *ADL-driven exploration, synthesis and validation of programmable architectures*

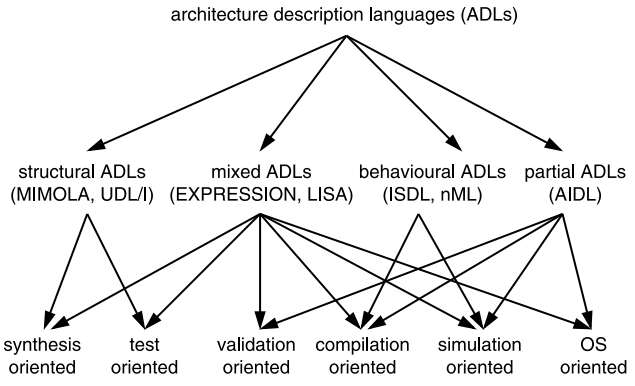

Fig. 4 *Taxonomy of ADLs*

synthesis-oriented, test-oriented, compilation-oriented, validation-oriented and operating system (OS) oriented.

ADLs can be classified into four categories based on the nature of the information: structural, behavioural, mixed and partial. The structural ADLs capture the structure in terms of architectural components and their connectivity. The behavioural ADLs capture the instruction set behaviour of the processor architecture. The mixed ADLs capture both structure and behaviour of the architecture. These ADLs capture complete description of the structure or behaviour or both. However, the partial ADLs capture specific information about the architecture for the intended task. For example, an ADL intended for interface synthesis does not require internal structure or behaviour of the processor.

Traditionally, structural ADLs are suitable for synthesis and test-generation. Similarly, behavioural ADLs are suitable for simulation and compilation. It is not always possible to establish a one-to-one correspondence between content-based and objective-based classification. For example, depending on the nature and amount of information captured, partial ADLs can represent any one or more classes of the objective-based ADLs. In this Section, we present the survey using content-based classification of ADLs.

3.1 Structural ADLs

ADL designers consider two important aspects: level of abstraction and generality. It is very difficult to find an abstraction to capture the features of different types of processors. A common way to obtain generality is to lower the abstraction level. Register transfer level (RT-level) is a popular abstraction level–low enough for detailed behaviour modelling of digital systems, and high enough to hide gate-level implementation details. Early ADLs are based on RT-level descriptions. In this Section, we briefly describe two structural ADLs: MIMOLA [6] and UDL/I [7].

3.1.1 MIMOLA: MIMOLA [6] is a structure-centric ADL developed at the University of Dortmund, Germany. It was originally proposed for micro-architecture design. One of the major advantages of MIMOLA is that the same description can be used for synthesis, simulation, test generation and compilation. A tool chain including the MSSH hardware synthesiser, the MSSQ code generator, the MSST self-test program compiler, the MSSB functional simulator, and the MSSU RT-level simulator were developed based on the MIMOLA language [6]. MIMOLA has also been used by the RECORD [8] compiler.

MIMOLA contains three parts: the algorithm to be compiled, the target processor model, and additional linkage and transformation rules. The software part (algorithm description) describes application programs in a PASCAL-like syntax. The processor model describes micro-architecture in the form of a component netlist. The linkage information is used by the compiler in order to locate important modules such as program counter and instruction memory. The following code segment specifies the program counter and instruction memory locations [6]:

```
LOCATION_FOR_PROGRAMCOUNTER PCReg;
LOCATION_FOR_INSTRUCTIONS IM[0..1023];
```

The algorithmic part of MIMOLA is an extension of PASCAL. Unlike other high level languages, it allows references to physical registers and memories. It also allows use of hardware components using procedure calls. For example, if the processor description contains a component named MAC, programmers can write the following code segment to use the multiply–accumulate operation performed by MAC:

```
res := MAC(x, y, z);
```

The processor is modelled as a net-list of component modules. MIMOLA permits modelling of arbitrary (programmable or nonprogrammable) hardware structures. Similar to VHDL, a number of predefined, primitive operators exists. The basic entities of MIMOLA hardware models are modules and connections. Each module is specified by its port interface and its behaviour. The following example shows the description of a multi-functional ALU module [6]:

```
MODULE ALU
    (IN inp1, inp2: (31:0);
     OUT outp: (31:0);
     IN ctrl: (1:0);
    )
CONBEGIN
      outp <- CASE ctrl OF
        0: inp1 + inp2 ;
        1: inp1 - inp2 ;
        2: inp1 AND inp2 ;
        3: inp1 ;
        END;
CONEND;
```

The CONBEGIN/CONEND construct includes a set of concurrent assignments. In the example a conditional assignment to output port *outp* is specified, which depends on the two-bit control input *ctrl*. The netlist structure is formed by connecting ports of module instances. For example, the following MIMOLA description connects two modules: *ALU* and accumulator *ACC*.

```
CONNECTIONS ALU.outp -> ACC.inp
            ACC.outp -> ALU.inp
```

The MSSQ code generator extracts instruction set information from the module netlist. It uses two internal data structures: connection operation graph (COG) and instruction tree (I-tree). It is a very difficult task to extract the COG and I-trees even in the presence of linkage information due to the flexibility of an RT-level structural description. Extra constraints need to be imposed in order for the MSSQ code generator to work properly. The constraints limit the architecture scope of MSSQ to micro-programmable controllers, in which all control signals originate directly from the instruction word. The lack of explicit description of processor pipelines or

resource conflicts may result in poor code quality for some classes of VLIW or deeply pipelined processors.

3.1.2 UDL/I: Unified design language, UDL/I, is [7] developed as a hardware description language for compiler generation in a COACH ASIP design environment at Kyushu University, Japan. UDL/I is used for describing processors at an RT-level on a per-cycle basis. The instruction set is automatically extracted from the UDL/I description [9], and then it is used for generation of a compiler and a simulator. COACH assumes simple RISC processors and does not explicitly support ILP or processor pipelines. The processor description is synthesisable with the UDL/I synthesis system [10]. The major advantage of the COACH system is that it requires a single description for synthesis, simulation, and compilation. The designer needs to provide hints to locate important machine states such as program counter and register files. Owing to difficulty in instruction set extraction (ISE), ISE is not supported for VLIW and superscalar architectures.

Structural ADLs enable flexible and precise micro-architecture descriptions. The same description can be used for hardware synthesis, test generation, simulation and compilation. However, it is difficult to extract the instruction set without restrictions on description style and target scope. Structural ADLs are more suitable for hardware generation than retargetable compilation.

3.2 Behavioural ADLs

The difficulty of instruction set extraction can be avoided by abstracting behavioural information from the structural details. Behavioural ADLs explicitly specify the instruction semantics and ignore detailed hardware structures. Typically, there is a one-to-one correspondence between behavioural ADLs and the instruction set reference manual. In this Section, we briefly describe two behavioural ADLs: nML [11] and ISDL [12].

3.2.1 nML: nML is an instruction set oriented ADL proposed at Technical University of Berlin, Germany. nML has been used by code generators CBC [13] and CHESS [14], and instruction set simulators Sigh/Sim [15] and CHECKERS. Currently, the CHESS/CHECKERS environment is used for automatic and efficient software compilation and instruction set simulation [16].

nML developers recognised the fact that several instructions share common properties. The final nML description would be compact and simple if the common properties are exploited. Consequently, nML designers used a hierarchical scheme to describe instruction sets. The instructions are the topmost elements in the hierarchy. The intermediate elements of the hierarchy are partial instructions (PI). The relationship between elements can be established using two composition rules: AND-rule and OR-rule. The AND-rule groups several PIs into a larger PI and the OR-rule enumerates a set of alternatives for one PI. Therefore instruction definitions in nML can be in the form of an and-or tree. Each possible derivation of the tree corresponds to an actual instruction.

To achieve the goal of sharing instruction descriptions, the instruction set is enumerated by an attributed grammar [17]. Each element in the hierarchy has a few attributes. A nonleaf element's attribute values can be computed based on its children's attribute values. Attribute grammar is also adopted by other ADLs such as ISDL [12] and TDL [18]. The following nML description shows an example of instruction specification [11]:

```
op numeric_instruction(a:num_action,
src:SRC, dst:DST)
action {
    temp_src = src;
    temp_dst = dst;
    a.action;
    dst = temp_dst;
}
op num_action = add | sub
op add()
action = {
  temp_dst = temp_dst + temp_src
}
```

The definition of *numeric_instruction* combines three partial instructions (PI) with the AND-rule: *num_action*, SRC and DST. The first PI, *num_action*, uses the OR-rule to describe the valid options for actions: *add* or *sub*. The number of all possible derivations of *numeric_instruction* is the product of the size of *num_action*, *SRC* and *DST*. The common behaviour of all these options is defined in the *action* attribute of *numeric_instruction*. Each option for *num_action* should have its own action attribute defined as its specific behaviour, which is referred by the *a.action* line. For example, the above code segment has an action description for *add* operation. Object code image and assembly syntax can also be specified in the same hierarchical manner.

nML also captures the structural information used by instruction set architecture (ISA). For example, storage units should be declared since they are visible to the instruction set. nML supports three types of storages: RAM, register and transitory storage. Transitory storage refers to machine states that are retained only for a limited number of cycles, e.g. values on buses and latches. Computations have no delay in the nML timing model – only storage units have delay. Instruction delay slots are modelled by introducing storage units as pipeline registers. The result of the computation is propagated through the registers in the behaviour specification.

nML models constraints between operations by enumerating all valid combinations. The enumeration of valid cases can make nML descriptions lengthy. More complicated constraints, which often appear in DSPs with irregular instruction level parallelism (ILP) constraints or VLIW processors with multiple issue slots, are hard to model with nML. For example, nML cannot model the constraint that operation *I1* cannot directly follow operation *I0*. nML explicitly supports several addressing modes. However, it implicitly assumes an architecture model which restricts its generality. As a result it is hard to model multi-cycle or pipelined units and multi-word instructions explicitly. A good critique of nML is given in [19].

3.2.2 ISDL: Instruction set description language (ISDL) was developed at MIT and used by the Aviv compiler [20] and GENSIM simulator generator [21]. The problem of constraint modelling is avoided by ISDL with explicit specification. ISDL is mainly targeted towards VLIW processors. Similar to nML, ISDL primarily describes the instruction set of processor architectures. ISDL consists of mainly five sections: instruction word format, global definitions, storage resources, assembly syntax and constraints. It also contains an optimisation information section that can be used to provide certain architecture specific hints for the compiler to make better machine dependent code optimisations.

The instruction word format section defines fields of the instruction word. The instruction word is separated into multiple fields each containing one or more subfields. The global definition section describes four main types: tokens, nonterminals, split functions and macro definitions. Tokens are the primitive operands of instructions. For each token, assembly format and binary encoding information must be defined. An example token definition of a binary operand is:

```
Token X[0..1] X_R ival {yylval.ival =
yytext[1] - '0';}
```

In this example, following the keyword *Token* is the assembly format of the operand. *X_R* is the symbolic name of the token used for reference. The *ival* is used to describe the value returned by the token. Finally, the last field describes the computation of the value. In this example, the assembly syntax allowed for the token *X_R* is *X0* or *X1*, and the values returned are 0 or 1, respectively.

The value (last) field is to be used for behavioural definition and binary encoding assignment by non-terminals or instructions. Nonterminal is a mechanism provided to exploit commonalities among operations. The following code segment describes a non-terminal named *XYSRC*:

```
Non_Terminal ival XYSRC: X_D {$$ = 0;} |
                         Y_D {$$ = Y_D + 1;};
```

The definition of *XYSRC* consists of the keyword *Non_Terminal*, the type of the returned value, a symbolic name as it appears in the assembly, and an action that describes the possible token or nonterminal combinations and the return value associated with each of them. In this example, *XYSRC* refers to tokens *X_D* and *Y_D* as its two options. The second field (*ival*) describes the returned value type. It returns 0 for *X_D* or incremented value for *Y_D*.

Similar to nML, storage resources are the only structural information modelled by ISDL. The storage section lists all storage resources visible to the programmer. It lists the names and sizes of the memory, register files and special registers. This information is used by the compiler to determine the available resources and how they should be used.

The assembly syntax section is divided into fields corresponding to the separate operations that can be performed in parallel within a single instruction. For each field, a list of alternative operations can be described. Each operation description consists of a name, a list of tokens or nonterminals as parameters, a set of commands that manipulate the bitfields, RTL description, timing details and costs. RTL description captures the effect of the operation on the storage resources. Multiple costs are allowed including operation execution time, code size, and costs due to resource conflicts. The timing model of ISDL describes when the various effects of the operation take place (e.g. because of pipelining).

In contrast to nML, which enumerates all valid combinations, ISDL defines invalid combinations in the form of Boolean expressions. This often leads to a simple constraint specification. It also enables ISDL to capture irregular ILP constraints. The following example shows how to describe the constraint that instruction I1 cannot directly follow instruction I0. The '[1]' indicates a time shift of one instruction fetch for the I0 instruction. The ' ~ ' is a symbol for NOT and '&' is for logical AND.

```
~ (I1 *) & ([1] I0 *, *)
```

ISDL provides the means for compact and hierarchical instruction set specification. However, it may not be possible to describe instruction sets with multiple encoding formats using the simple tree-like instruction structure of ISDL.

In general, the behavioural languages have one feature in common: hierarchical instruction set description based on attribute grammar [17]. This feature simplifies the instruction set description by sharing the common components between operations. However, the capabilities of these models are limited due to the lack of detailed pipeline and timing information. It is not possible to generate cycle accurate simulators without certain assumptions regarding control behaviour. Due to lack of structural details, it is also not possible to perform resource-based scheduling using behavioural ADLs.

3.3 Mixed ADLs

Mixed languages capture both structural and behavioural details of the architecture. In this Section, we briefly describe three mixed ADLs: HMDES, EXPRESSION and LISA.

3.3.1 HMDES: Machine description language HMDES was developed at University of Illinois at Urbana-Champaign for the IMPACT research compiler [22]. C-like preprocessing capabilities such as file inclusion, macro expansion and conditional inclusion are supported in HMDES. An HMDES description is the input to the MDES machine description system of the Trimaran compiler infrastructure, which contains IMPACT as well as the Elcor research compiler from HP Labs. The description is first pre-processed, then optimised and translated to a low-level representation file. A machine database reads the low level files and supplies information for the compiler back end through a predefined query interface.

MDES captures both structure and behaviour of target processors. Information is broken down into sections such as format, resource usage, latency, operation and register. For example, the following code segment describes register and register file. It describes 64 registers. The register file describes the width of each register and other optional fields such as generic register type (virtual field), speculative, static and rotating registers. The value '1' implies speculative and '0' implies nonspeculative.

```
SECTION Register {
  R0(); R1();… R63();
  'R[0]'();… 'R[63]'();
  …
}

SECTION Register_ File {
  RF_i(width(32) virtual(i) speculative(1)
    static(R0…R63)rotating('R[0]'…'R[63]' ));
  …
}
```

MDES allows only a restricted retargetability of the cycle-accurate simulator to the HPL-PD processor family [23]. MDES permits description of memory systems, but limited to the traditional hierarchy, i.e. register files, caches and main memory.

3.3.2 EXPRESSION: The above ADLs require explicit description of reservation tables (RT). Processors that contain complex pipelines, large amounts of parallelism and complex storage sub-systems typically contain a large

number of operations and resources (and hence RTs). Manual specification of RTs on a per-operation basis thus becomes cumbersome and error-prone. The manual specification of RTs (for each configuration) becomes impractical during rapid architectural exploration. The EXPRESSION ADL [24] describes a processor as a netlist of units and storages to automatically generate RTs based on the netlist [25]. Unlike MIMOLA, the netlist representation of EXPRESSION is coarse grain. It uses a higher level of abstraction similar to the block-diagram level description in architecture manual.

EXPRESSION ADL was developed at University of California, Irvine. The ADL has been used by the retargetable compiler (EXPRESS [26]) and simulator (SIMPRESS [27]) generation framework. The framework also supports a graphical user interface (GUI) and can be used for design space exploration of programmable architectures consisting of processor cores, coprocessors and memories.

An EXPRESSION description is composed of two main sections: behaviour (instruction set) and structure. The behaviour section has three subsections: operations, instruction, and operation mappings. Similarly, the structure section consists of three subsections: components, pipeline/data–transfer paths and memory subsystem.

The operation subsection describes the instruction set of the processor. Each operation of the processor is described in terms of its opcode and operands. The types and possible destinations of each operand are also specified. A useful feature of EXPRESSION is an operation group that groups similar operations together for ease of later reference. For example, the following code segment shows an operation group (*alu_ops*) containing two ALU operations: *add* and *sub*.

```
(OP_GROUP alu_ops
  (OPCODE add
    (OPERANDS (SRC1 reg) (SRC2 reg/imm)
    (DEST reg))
    (BEHAVIOUR DEST = SRC1 + SRC2)
    …
  )
  (OPCODE sub
    (OPERANDS (SRC1 reg) (SRC2 reg/imm)
    (DEST reg))
    (BEHAVIOUR DEST = SRC1 − SRC2)
    …
  )
)
```

The instruction subsection captures the parallelism available in the architecture. Each instruction contains a list of slots (to be filled with operations), with each slot corresponding to a functional unit. The operation mapping subsection is used to specify the information needed by instruction selection and architecture-specific optimisations of the compiler. For example, it contains mapping between generic and target instructions.

The component subsection describes each RT-level component in the architecture. The components can be pipeline units, functional units, storage elements, ports and connections. For multi-cycle or pipelined units, the timing behaviour is also specified.

The pipeline/data-transfer path subsection describes the netlist of the processor. The *pipeline path description* provides a mechanism to specify the units which comprise the pipeline stages, while the *data-transfer path description* provides a mechanism for specifying the valid data-transfers. This information is used to both retarget the simulator and to generate reservation tables needed by the scheduler [25]. An example path declaration for the DLX architecture [28] (Fig. 5) is shown below. It describes that the processor has five pipeline stages. It also describes that the *Execute* stage has four parallel paths. Finally, it describes each path, e.g. it describes that the *FADD* path has four pipeline stages.

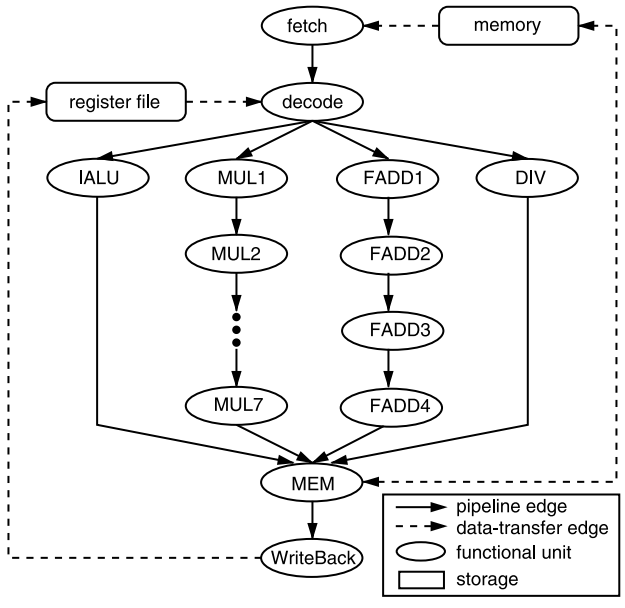

Fig. 5 *DLX architecture*

```
(PIPELINE Fetch Decode Execute MEM Write-
Back)
(Execute (ALTERNATE IALU MULT FADD DIV))
(MULT (PIPELINE MUL1 MUL2… MUL7))
(FADD (PIPELINE FADD1 FADD2 FADD3 FADD4))
```

The memory subsection describes the types and attributes of various storage components (such as register files, SRAMs, DRAMs and caches). The memory netlist information can be used to generate memory aware compilers and simulators. Memory aware compilers can exploit the detailed information to hide the latency of the lengthy memory operations [29].

In general, EXPRESSION captures the data path information in the processor. The control path is not explicitly modelled. Also, the VLIW instruction composition model is simple. The instruction model requires extension to capture inter-operation constraints such as sharing of common fields. Such constraints can be modeled by ISDL through cross-field encoding assignment.

3.3.3 LISA: LISA (language for instruction set architecture) [30] was developed at Aachen University of Technology, Germany, with a simulator centric view. The language has been used to produce production quality simulators [31]. An important aspect of the LISA language is its ability to capture control paths explicitly. Explicit modelling of both datapath and control is necessary for cycle-accurate simulation. LISA has also been used to generate parts of a compiler's instruction scheduler [32].

LISA descriptions are composed of two types of declarations: resource and operation. The resource declarations cover hardware resources such as registers, pipelines and memories. The pipeline model defines all possible

pipeline paths that operations can go through. An example pipeline description for the architecture shown in Fig. 5 is as follows:

```
PIPELINE pipe_int = {Fetch; Decode; IALU;
MEM; WriteBack }
PIPELINE pipe_float = {Fetch; Decode; FADD1;
FADD2; FADD3; FADD4; MEM; WriteBack }
PIPELINE pipe_mul = {Fetch; Decode; MUL1;
MUL2;... MUL7; MEM; WriteBack }
PIPELINE pipe_div = {Fetch; Decode; DIV;
MEM; WriteBack}
```

Operations are the basic objects in LISA. They represent the designer's view of the behaviour, the structure and the instruction set of the programmable architecture. Operation definitions capture the description of different properties of the system such as operation behaviour, instruction set information and timing. These operation attributes are defined in several sections:

- The CODING section describes the binary image of the instruction word.
- The SYNTAX section describes the assembly syntax of instructions.
- The SEMANTICS section specifies the instruction set semantics.
- The BEHAVIOUR section describes components of the behavioural model.
- The ACTIVATION section describes the timing of other operations relative to the current operation.
- The DECLARE section contains local declarations of identifiers.

LISA exploits the commonality of similar operations by grouping them into one. The following code segment describes the decoding behaviour of two immediate-type (i_type) operations (ADDI and SUBI) in the DLX *Decode* stage. The complete behaviour of an operation can be obtained by combining its behaviour definitions in all the pipeline stages.

```
OPERATION i_type IN pipe_int.Decode {
  DECLARE {
    GROUP opcode = {ADDI || SUBI}
    GROUP rs1, rd = {fix_register};
  }
  CODING {opcode rs1 rd immediate}
  SYNTAX {opcode rd "," rs1 "," immediate}
  BEHAVIOUR {reg_a = rs1; imm = immediate;
cond = 0;
  }
  ACTIVATION {opcode, writeback}
}
```

A language similar to LISA is RADL. RADL [33] was developed at Rockwell, Inc. as an extension of the LISA approach that focuses on explicit support of detailed pipeline behaviour to enable generation of production quality cycle-accurate and phase-accurate simulators.

3.4 Partial ADLs

The ADLs discussed so far capture the complete description of the processor's structure, behaviour or both. There are many description languages that capture partial information of the architecture needed to perform specific tasks. In this Section, we describe two such ADLs.

AIDL is an ADL developed at University of Tsukuba for design of high-performance superscalar processors [34]. It seems that AIDL does not aim at datapath optimisation but aims at validation of the pipeline behaviour such as data-forwarding and out-of-order completion. In AIDL the timing behaviour of pipeline is described using interval temporal logic. AIDL does not support software toolkit generation. However, AIDL descriptions can be simulated using the AIDL simulator.

PEAS-I is a CAD system for ASIP design supporting automatic instruction set optimisation, compiler generation, and instruction level simulator generation [35]. In the PEAS-I system, the GNU C compiler is used, and the machine description of GCC is automatically generated. Therefore, there exists no specific ADL in PEAS-I. Inputs to PEAS-I include an application program written in C and input data to the program. Then, the instruction set is automatically selected in such a way that the performance is maximised or the gate count is minimised. Based on the instruction set, GNU CC and an instruction level simulator are automatically retargeted.

4 ADL driven methodologies

The survey of ADLs is incomplete without a clear understanding of the supported methodologies. In this Section, we investigate the contribution of the contemporary ADLs in the following methodologies:

- software toolkit generation and exploration
- generation of hardware implementation
- top-down validation.

4.1 Toolkit generation and exploration

Embedded systems present a tremendous opportunity to customise designs by exploiting the application behaviour. Rapid exploration and evaluation of candidate architectures are necessary due to time-to-market pressure and short product lifetimes. ADLs are used to specify processor and memory architectures and generate a software toolkit including compiler, simulator, assembler, profiler and debugger. Figure. 6 shows a traditional ADL-based design space exploration flow. The application programs are compiled and simulated, and the feedback is used to modify the ADL specification with the goal of finding the best possible architecture for the given set of application programs under various design constraints such as area, power and performance.

An extensive body of recent work addresses ADL driven software toolkit generation and design space exploration of processor-based embedded systems, in both academia: ISDL [12], Valen-C [36], MIMOLA [8], LISA [30], nML [11], Sim-nML [37], EXPRESSION [24]; and industry:

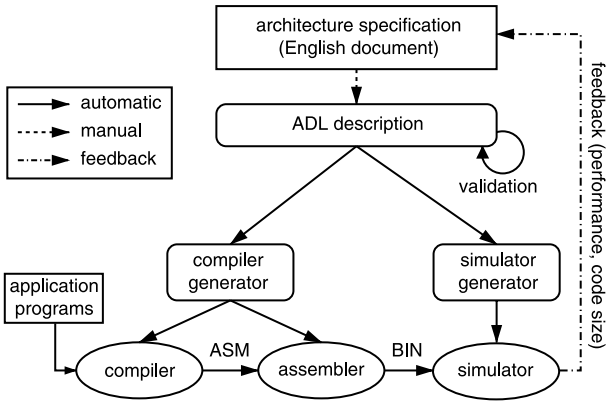

Fig. 6 *ADL-driven design space exploration*

ARC [38], Axys [39], RADL [33], Target [16], Tensilica [40], MDES [23].

One of the main purposes of an ADL is to support automatic generation of a high-quality software toolkit including at least an ILP (instruction level parallelism) compiler and a cycle-accurate simulator. However, such tools require detailed information about the processor, typically in a form that is not concise and easily specifiable. Therefore, it becomes necessary to develop procedures to automatically generate such tool-specific information from the ADL specification. For example, reservation tables (RTs) are used in many ILP compilers to describe resource conflicts. However, manual description of RTs on a per-instruction basis is cumbersome and error-prone. Instead, it is easier to specify the pipeline and datapath resources in an abstract manner, and generate RTs on a per-instruction basis [25].

In this Section we describe some of the challenges in automatic generation of software tools (focusing on compilers and simulators) and survey some of the approaches adopted by current tools.

4.1.1 Compilers: Traditionally, software for embedded systems was hand-tuned in assembly. With increasing complexity of embedded systems, it is no longer practical to develop software in assembly language or to optimise it manually except for critical sections of the code. Compilers which produce optimised machine specific code from a program specified in a high-level language (HLL) such as C/C++ and Java are necessary to produce efficient software within the time budget. Compilers for embedded systems have been the focus of several research efforts recently [41].

The compilation process can be broadly broken into two steps: analysis and synthesis [42]. During analysis, the program (in HLL) is converted into an intermediate representation (IR) that contains all the desired information such as control and data dependences. During synthesis, the IR is transformed and optimised in order to generate efficient target specific code. Traditionally, compilers have been painstakingly hand-tuned to a particular architecture (or architecture class) and application domain(s) for generating optimised code. However, stringent time-to-market constraints for SOC designs no longer make it feasible to manually generate compilers tuned to particular architectures. Automatic generation of an efficient compiler from an abstract description of the processor model becomes essential.

A promising approach to automatic compiler generation is the 'retargetable compiler' approach. A compiler is classified as retargetable if it can be adapted to generate code for different target processors with significant reuse of the compiler source code. Retargetability is typically achieved by providing target machine information (in an ADL) as input to the compiler along with the program corresponding to the application.

The complexity in retargeting the compiler depends on the range of target processors it supports and also on its optimising capability. Owing to the growing number of instruction level parallelism (ILP) features in modern processor architectures, the difference in quality of code generated by a naive code conversion process and an optimising ILP compiler can be enormous. Recent approaches on retargetable compilation have focused on developing optimisations/transformations that are 'retargetable' and capturing the machine specific information needed by such optimisations in the ADL. We classify retargetable compilers into three broad categories, based on the type of the machine model accepted as input.

(i) *Architecture template based*: Such compilers assume a limited architecture template which is parameterisable for customisation. The most common parameters include operation latencies, number of functional units, number of registers, etc. Architecture template based compilers have the advantage that both optimisations and the phase ordering between them can be manually tuned to produce highly efficient code for the limited architecture space. Examples of such compilers include the Valen-C compiler [36] and the GNU-based C/C++ compiler from Tensilica Inc. [40]. The Tensilica GNU-based C/C++ compiler is geared towards the Xtensa parameterisable processor architecture. One important feature of this system is the ability to add new instructions (described through an instruction extension language) and automatically generate software tools tuned to the new instruction set.

(ii) *Explicit behavioural information based*: Most compilers require a specification of the behaviour to retarget their transformations (e.g. instruction selection requires a description of the semantics of each operation). Explicit behavioural information based retargetable compilers require full information about the instruction set as well as explicit resource conflict information. Examples include the AVIV [20] compiler using ISDL, CHESS [14] using nML, and Elcor [23] using MDes. The AVIV retargetable code generator produces machine code, optimised for minimal size, for target processors with different instruction sets. It solves the phase ordering problem by performing a heuristic branch-and-bound step that performs resource allocation/assignment, operation grouping, and scheduling concurrently. CHESS is a retargetable code generation environment for fixed-point DSP processors. CHESS performs instruction selection, register allocation and scheduling as separate phases (in that order). Elcor is a retargetable compilation environment for VLIW architectures with speculative execution. It implements a software pipelining algorithm (modulo scheduling) and register allocation for static and rotating register files.

(iii) *Behavioural information generation based*: Recognising that the architecture information needed by the compiler is not always in a form that may be well suited for other tools (such as synthesis) or does not permit concise specification, some research has focused on extraction of such information from a more amenable specification. Examples include the MSSQ and RECORD compiler using MIMOLA [8], the retargetable C compiler based on LISA [43], and the EXPRESS compiler using EXPRESSION [24]. MSSQ translates Pascal-like high-level language (HLL) into microcode for micro-programmable controllers, while RECORD translates code written in a DSP-specific programming language, called data flow language (DFL), into machine code for the target DSP. The retargetable C compiler generation using LISA is based on reuse of a powerful C compiler platform with many built-in code optimisations and generation of mapping rules for code selection using the instruction semantics information [43]. The EXPRESS compiler tries to bridge the gap between explicit specification of all information (e.g. AVIV) and implicit specification requiring extraction of instruction set (e.g. RECORD), by having a mixed behavioural/structural view of the processor.

4.1.2 Simulators: Simulators are critical components of the exploration and software design toolkit for the system designer. They can be used to perform diverse tasks such as verifying the functionality and/or timing

behaviour of the system (including hardware and software), and generating quantitative measurements (e.g. power consumption) which can be used to aid the design process.

Simulation of the processor system can be performed at various abstraction levels. At the highest level of abstraction, a functional simulation of the processor can be performed by modelling only the instruction set (IS). Such simulators are termed instruction set simulators (ISS) or instruction-level simulators (ILS). At lower levels of abstraction are the cycle-accurate and phase-accurate simulation models that yield more detailed timing information. Simulators can be further classified based on whether they provide bit-accurate models, pin-accurate models, exact pipeline models and structural models of the processor.

The model of simulation adopted has a significant impact on the simulation speed and flexibility of the simulator. Based on the simulation model, simulators can be classified into three types: interpretive, compiled and mixed.

(i) *Interpretation based*: Such simulators are based on an interpretive model of the processor's instruction set. Interpretive simulators store the state of the target processor in host memory. Then follows a fetch, decode and execute model: instructions are fetched from memory, decoded and then executed in serial order. Advantages of this model include ease of implementation, flexibility and the ability to collect varied processor state information. However, it suffers from significant performance degradation as compared to the other approaches, primarily due to the tremendous overhead in fetching, decoding and dispatching instructions. Almost all commercially available simulators are interpretive. Examples of research interpretive retargetable simulators include SIMPRESS [27] using EXPRESSION and GENSIM/XSIM [21] using ISDL.

(ii) *Compilation based*: Compilation based approaches reduce the runtime overhead by translating each target instruction into a series of host machine instructions which manipulate the simulated machine state. Such translation can be done either at compile time (static compiled simulation), where the fetch-decode-dispatch overhead is completely eliminated, or at load time (dynamic compiled simulation), which amortises the overhead over repeated execution of code. Simulators based on the static compilation model are presented by Zhu and Gajski [44] and Pees *et al.* [31]. Examples of dynamic compiled code simulators include the Shade simulator [45], and the Embra simulator [46].

(iii) *Interpretive+compiled*: Traditional interpretive simulation is flexible but slow. Instruction decoding is a time-consuming process in a software simulation. Compiled simulation performs compile-time decoding of application programs to improve the simulation performance. However, all compiled simulators rely on the assumption that the complete program code is known before the simulation starts and is furthermore runtime static. Owing to the restrictiveness of the compiled technique, interpretive simulators are typically used in embedded systems design flow. Two recently proposed simulation techniques (JIT-CCS [47] and IS-CS [48]) combine the flexibility of interpretive simulation with the speed of compiled simulation.

4.2 Generation of Hardware Implementation

Recent approaches on ADL-based software toolkit generation enable performance driven exploration. The simulator produces profiling data and thus may answer questions concerning the instruction set, the performance of an algorithm and the required size of memory and registers. However, the required silicon area, clock frequency and power consumption can only be determined in conjunction with a synthesisable HDL model.

There are two major approaches in the literature for synthesisable HDL generation. The first one is a parameterised processor core based approach. These cores are bound to a single processor template whose architecture and tools can be modified to a certain degree. The second approach is based on processor specification languages.

Examples of processor template based approaches are Xtensa [40], Jazz [49] and PEAS [50]. Xtensa [40] is a scalable RISC processor core. Configuration options include the width of the register set, caches and memories. New functional units and instructions can be added using the Tensilica Instruction Language (TIE). A synthesisable hardware model along with software toolkit can be generated for this class of architecture. Improv's Jazz [49] processor is supported by a flexible design methodology to customise the computational resources and instruction set of the processor. It allows modifications of data width, number of registers, depth of hardware task queue, and addition of custom functionality in Verilog. PEAS [50] is a GUI based hardware/software codesign framework. It generates HDL code along with software toolkit. It has support for several architecture types and a library of configurable resources.

Figure 7 shows a typical framework of processor description language driven HDL generation. Structure-centric ADLs such as MIMOLA are suitable for hardware generation. Some of the behavioural languages (such as ISDL and nML) are also used for hardware generation. For example, the HDL generator HGEN [51] uses ISDL description, and the synthesis tool GO [16] is based on nML. Itoh *et al.* [52] have proposed a micro-operation description based synthesisable HDL generation. It can handle simple processor models with no hardware interlock mechanism or multi-cycle operations.

Mixed languages such as LISA and EXPRESSION capture both the structure and behaviour of the processor. The synthesisable HDL generation approach based on the LISA language [53] produces an HDL model of the architecture. The designer has the choice to generate a VHDL, Verilog or SystemC representation of the target architecture [54]. The HDL generation methodology presented by Mishra *et al.* [55, 56] combines the advantages of the processor template based environments and the language based specifications using EXPRESSION ADL.

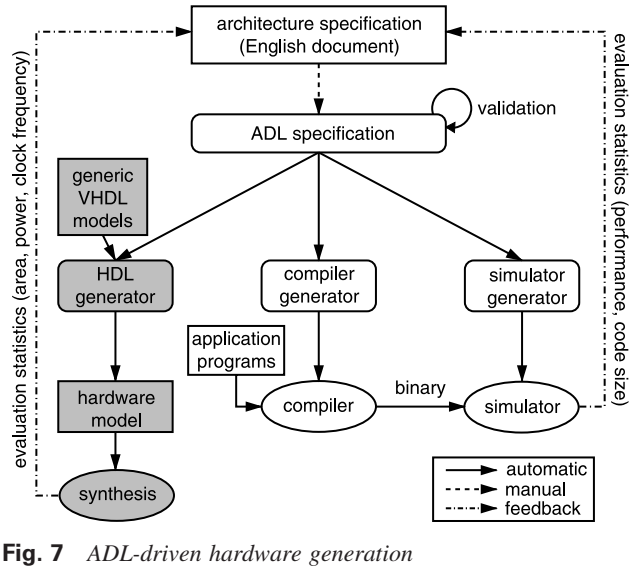

Fig. 7 *ADL-driven hardware generation*

4.3 Top-down validation

Validation of microprocessors is one of the most complex and important tasks in the current system-on-chip (SOC) design methodology. Figure. 8 shows a traditional architecture validation flow. The architect prepares an informal specification of the microprocessor in the form of an English document. The logic designer implements the modules in the register-transfer level (RTL). The *RTL design* is validated using a combination of simulation techniques and formal methods. One of the most important problems in today's processor design validation is the lack of a golden reference model that can be used for verifying the design at different levels of abstraction. Thus, many existing validation techniques employ a *bottom-up approach* to pipeline verification, where the functionality of an existing pipelined processor is, in essence, reverse-engineered from its RT-level implementation.

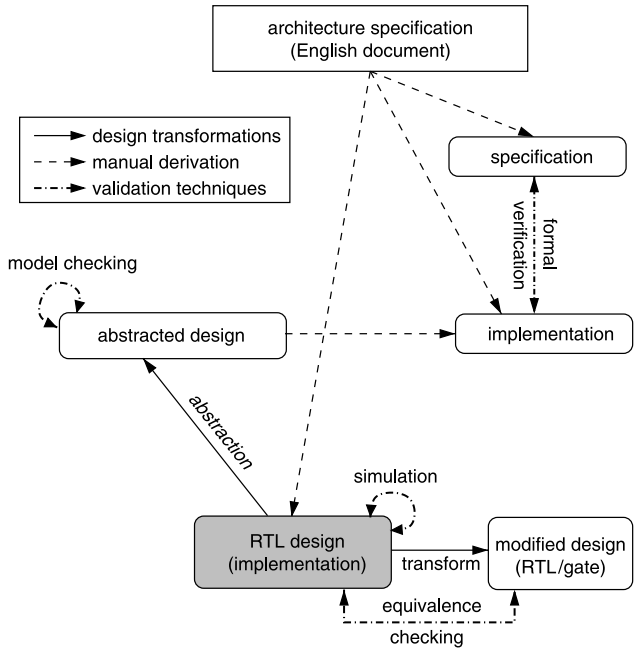

Fig. 8 *Bottom-up validation flow*

Mishra [57] has presented an ADL-driven validation technique that is complementary to these bottom-up approaches. It leverages the system architect's knowledge about the behaviour of the programmable embedded systems through ADL constructs, thereby allowing a powerful *top-down approach* to microprocessor validation. Figure 9 shows an ADL-driven top-down validation methodology. This methodology has two important steps: validation of ADL specification, and specification-driven validation of programmable architectures.

4.3.1 Validation of ADL specification: It is important to verify the ADL specification to ensure the correctness of the architecture specified and the generated software toolkit. Both static and dynamic behaviour need to be verified to ensure that the specified architecture is well formed. The static behaviour can be validated by analysing several static properties, such as connectedness, false pipeline and data-transfer paths and completeness using a graph based model of the pipelined architecture [58–60].

The dynamic behaviour can be validated by analysing the instruction flow in the pipeline using a finite state machine (FSM) based model to verify several important architectural properties such as determinism and in-order execution in the presence of hazards and multiple exceptions [61–63].

4.3.2 Specification-driven validation: The validated ADL specification can be used as a golden reference model for top-down validation of programmable architectures. The top-down validation approach has been demonstrated in two directions: functional test program generation, and design validation using a combination of equivalence checking and symbolic simulation.

Test generation for functional validation of processors has been demonstrated using MIMOLA [6], EXPRESSION [64], and nML [16]. A model checking based approach is used to automatically generate functional test programs from the processor specification using EXPRESSION ADL [64, 65]. It generates a graph model of the pipelined processor from the ADL specification. The functional test programs are generated based on the coverage of the pipeline behaviour.

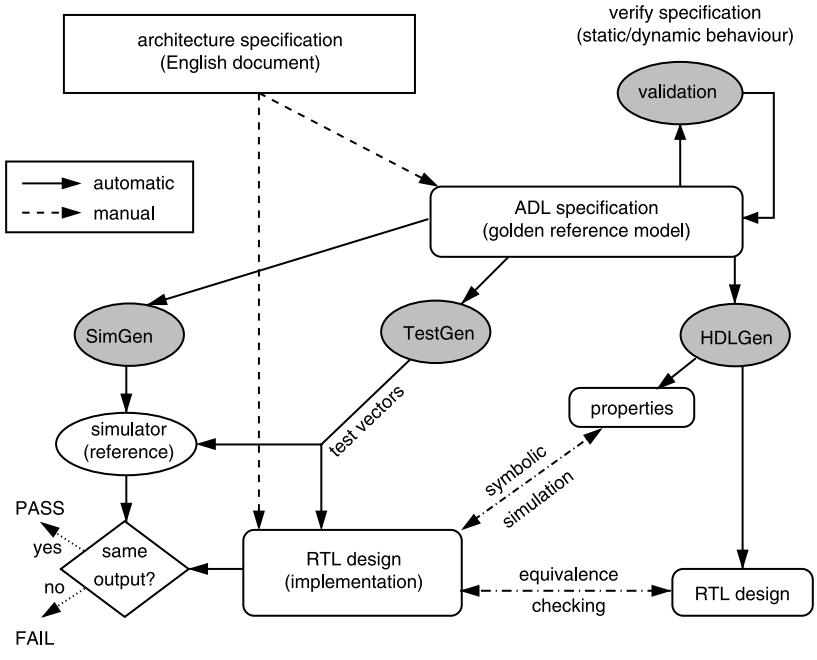

Fig. 9 *Top-down validation methodology*

ADL-driven design validation using equivalence checking has been demonstrated using EXPRESSION ADL [66]. This approach combines ADL-driven hardware generation and validation. The generated hardware model (RTL) is used as a reference model to verify the handwritten implementation (RTL design) of the processor. To verify that the implementation satisfies certain properties, the framework generates the intended properties. These properties are applied using symbolic simulation [66].

5 Comparative study

Table 1 compares the features of contemporary ADLs in terms of their support for compiler generation, simulator generation, test generation, synthesis and formal verification. Also, information captured by the ADLs is compared.

Since MIMOLA and UDL/I are originally HDLs, their descriptions are synthesisable and can be simulated using HDL simulators. MIMOLA appears to be successful for retargetable compilation for DSPs with irregular datapaths. However, since its abstraction level is rather low, MIMOLA is laborious to write. COACH (uses UDL/I) supports generation of both compilers and simulators. nML and ISDL support ILP compiler generation. However, owing to the lack of structural information, it is not possible to automatically detect resource conflicts between instructions. MDES supports simulator generation only for the HPL-PD processor family. EXPRESSION has the ability to automatically generate ILP compilers, reservation tables and cycle-accurate simulators. Furthermore, description of memory hierarchies is supported. LISA and RADL were originally designed for simulator generation. AIDL descriptions are executable on the AIDL simulator and do not support compiler generation.

From the above comparison, as well as our experiences of research/development, we believe that ADLs should capture both behaviour (instruction set) and structure (net-list) information to generate high-quality software toolkit automatically and efficiently. Behaviour information which is necessary for compiler generation should be explicitly specified for mainly two reasons. First, instruction set extraction from net-lists described in synthesis-oriented ADLs or HDLs does not seem to be applicable to a wide range of processors. Second, synthesis-oriented ADLs or HDLs are generally tedious to write for the purpose of DSE. Also, structure information is necessary not only to generate cycle-accurate simulators but also to generate ILP constraints which are necessary for high-quality ILP compiler generation.

ADLs designed for a specific domain (such as DSP or VLIW) or for a specific purpose (such as simulation or compilation) can be compact and it is possible to automatically generate efficient (in terms of area, time and power) tools/hardwares. However, it is difficult to design an ADL for a wide variety of architectures to perform different tasks using the same specification. Generic ADLs require the support of powerful methodologies to generate high quality results compared to domain-specific/task-specific ADLs.

6 Conclusions

In the past, an ADL was designed to serve a specific purpose. For example, MIMOLA and UDL have features similar to a hardware description language and were used mainly for synthesis of processor architectures. Similarly, LISA and RADL were designed for simulation of processor architectures. Likewise, MDES and EXPRESSION were designed mainly for generating retargetable compilers.

The early ADLs were either structure-oriented (MIMOLA, UDL/I), or behaviour-oriented (nML, ISDL). As a result, each class of ADLs is suitable for specific tasks. For example, structure-oriented ADLs is suitable for hardware synthesis and unfit for compiler generation. Similarly, behaviour-oriented ADLs are appropriate for generating compiler and simulator for instruction set architectures, and unsuited for generating cycle-accurate simulator or hardware implementation of the architecture. The later ADLs (LISA and EXPRESSION) adopted the mixed approach where the language captures both the structure and behaviour of the architecture.

At present, the existing ADLs are getting modified with the new features and methodologies to perform software toolkit generation, synthesis, and test generation for validation of architectures. For example, nML is extended by Target Compiler Technologies [16] to perform synthesis and test generation. Similarly, the LISA language has been used for hardware generation [54, 67], instruction encoding synthesis [68] and JTAG interface generation [69]. Likewise, EXPRESSION has been used for hardware generation [56], instruction set synthesis [70], test generation [64, 71], and specification validation [58, 62].

The majority of the ADLs were designed mainly for processor architectures. MDES has features for specifying both processor and memory architectures. EXPRESSION allows specification of processor, memory and co-processor architectures [72]. Similarly, the language elements of LISA

Table 1: Comparison between different ADLs

	MIMOLA	UDL/I	nML	ISDL	HMDES	EXPRESSION	LISA	RADL	AIDL
Compiler generation	√	√	√	√	√	√	√		
Simulator generation	√	√	√	√	√	√	√	√	√
Cycle-accurate simulation	√	△		√	√	√	√	√	△
Formal verification						△			△
Hardware generation	√	√		△		√	√		△
Test generation	√		√			√			
JTAG interface generation							√		
Instruction set information			√	√	√	√	√	√	
Structural information	√	√			√	√	√	√	
Memory information					△	√	√		

√ supported; △ supported with restrictions

enable the description of processor, memory, peripherals and external interfaces [69, 73].

In the future, the existing ADLs will go through changes in two dimensions. First, ADLs will specify not only processor, memory and co-processor architectures but also other components of the system-on-chip architectures including peripherals and external interfaces. Second, ADLs will be used for software toolkit generation, hardware synthesis, test generation, instruction set synthesis, and validation of microprocessors. Furthermore, multiprocessor SOCs will be captured and various attendant tasks will be addressed. The tasks include support for formal analysis, generation of real-time operating systems (RTOS), exploration of communication architectures, and support for interface synthesis. The emerging ADLs will have these features.

7 Acknowledgments

This work was partially supported by NSF grants CCR-0203813 and CCR-0205712. We thank Prof. Alex Nicolau and members of the ACES laboratory for their helpful comments and suggestions. We also thank Mr. Anupam Chattopadhyay for his constructive comments and feedback.

8 References

1 Qin, W., and Malik, S.: 'Architecture description languages for retargetable compilation', in 'The compiler design handbook: optimizations machine code generation' (CRC Press, 2002)
2 Tomiyama, H., Halambi, A., Grun, P., Dutt, N., and Nicolau, A.: 'Architecture description languages for systems-on-chip design'. Proc. Asia Pacific Conf. on Chip Design Language, 1999, pp. 109–116
3 Clements, P.C.: 'A survey of architecture description languages'. Proc. Int. Workshop on Software Specification and Design (IWSSD), 1996, pp. 16–25
4 Medvidovic, N., and Taylor, R.: 'A framework for classifying and comparing architecture description languages'. Proc. Eur. Software Engineering Conf. (ESEC), Springer–Verlag, 1997, pp. 60–76
5 Barbacci, M.R.: 'Instruction set processor specifications (isps): The notation and its applications', *IEEE Trans. Comput.*, 1981, **30**, (1), pp. 24–40
6 Leupers, R., and Marwedel, P.: 'Retargetable code generation based on structural processor descriptions', *Des. Autom. Embedded Syst.*, 1998, **3**, (1), pp. 75–108
7 Akaboshi, H.: 'A study on design support for computer architecture design'. PhD thesis, Dept. of Information Systems, Kyushu University, Japan, Jan 1996
8 Leupers, R., and Marwedel, P.: 'Retargetable generation of code selectors from HDL processor models'. Proc. Eur. Des. Test Conf., 1997, pp. 140–144
9 Akaboshi, H., and Yasuura, H.: 'Behaviour extraction of MPU from HDL description'. Proc. Asia Pacific Conf. on Hardware Description Languages (APCHDL), 1994
10 http://pjro.metsa.astem.or.jp/udli. UDL/I Simulation/Synthesis Environment, 1997
11 Freericks, M.: 'The nML machine description formalism'. Technical Report TR SM-IMP/DIST/08, TU Berlin, CS Dept., 1993
12 Hadjiyiannis, G., Hanono, S., and Devadas, S.: 'ISDL: An instruction set description language for retargetability'. Proc. Design Automation Conf. (DAC), 1997, pp. 299–302
13 Fauth, A., and Knoll, A.: 'Automatic generation of DSP program development tools'. Proc. Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP), 1993, pp. 457–460
14 Lanneer, D., Praet, J., Kifli, A., Schoofs, K., Geurts, W., Thoen, F., and Goossens, G.: 'CHESS: Retargetable code generation for embedded DSP processors', in 'Code generation for embedded processors' (Kluwer Academic Publishers, 1995), pp. 85–102
15 Lohr, F., Fauth, A., and Freericks, M.: Sigh/sim: An environment for retargetable instruction set simulation. Technical Report 1993/43, Dept. Computer Science, Tech. Univ. Berlin, Germany, 1993
16 http://www.retarget.com. Target Compiler Technologies
17 Paakki, J.: 'Attribute grammar paradigms – a high level methodology in language implementation', *ACM Comput. Surv.*, 1995, **27**, (2), pp. 196–256
18 Kastner, D.: Tdl: A hardware and assembly description languages. Technical Report TDL 1.4, Saarland University, Germany, 2000
19 Hartoog, M., Rowson, J., Reddy, P., Desai, S., Dunlop, D., Harcourt, E., and Khullar, N.: 'Generation of software tools from processor descriptions for hardware/software codesign'. Proc. Design Automation Conf. (DAC), 1997, pp. 303–306
20 Hanono, S., and Devadas, S.: 'Instruction selection, resource allocation, and scheduling in the AVIV retargetable code generator'. Proc. Design Automation Conf. (DAC), 1998, pp. 510–515
21 Hadjiyiannis, G., Russo, P., and Devadas, S.: 'A methodology for accurate performance evaluation in architecture exploration'. Proc. Design Automation Conf. (DAC), 1999, pp. 927–932
22 Gyllenhaal, J., Rau, B., and Hwu, W.: Hmdes version 2.0 specification. Technical Report IMPACT-96-3, IMPACT Research Group, Univ. of Illinois, Urbana, IL, 1996
23 The MDES User Manual. http://www.trimaran.org. 1997
24 Halambi, A., Grun, P., Ganesh, V., Khare, A., Dutt, N., and Nicolau, A.: 'EXPRESSION: A language for architecture exploration through compiler/simulator retargetability'. Proc. Design Automation and Test in Europe (DATE), 1999, pp. 485–490
25 Grun, P., Halambi, A., Dutt, N., and Nicolau, A.: 'RTGEN: An algorithm for automatic generation of reservation tables from architectural descriptions'. Proc. Int. Symp. on System Synthesis (ISSS), 1999, pp. 44–50
26 Halambi, A., Shrivastava, A., Dutt, N., and Nicolau, A.: 'A customizable compiler framework for embedded systems'. Proc. Software and Compilers for Embedded Systems (SCOPES), 2001
27 Khare, A., Savoiu, N., Halambi, A., Grun, P., Dutt, N., and Nicolau, A.: 'V-SAT: A visual specification and analysis tool for system-on-chip exploration'. Proc. EUROMICRO Conf., 1999, pp. 1196–1203
28 Hennessy, J., and Patterson, D.: 'Computer architecture: a quantitative approach' (Morgan Kaufmann Publishers Inc, San Mateo, CA, 1990)
29 Grun, P., Dutt, N., and Nicolau, A.: 'Memory aware compilation through accurate timing extraction'. Proc. Design Automation Conf. (DAC), 2000, pp. 316–321
30 Zivojnovic, V., Pees, S., and Meyr, H.: 'LISA - machine description language and generic machine model for HW/SW co-design'. IEEE Workshop on VLSI Signal Processing, 1996, pp. 127–136
31 Pees, S., Hoffmann, A., and Meyr, H.: 'Retargetable compiled simulation of embedded processors using a machine description language', *ACM Trans. Des. Autom. Electron. Syst.*, 2000, **5**, (4), pp. 815–834
32 Wahlen, O., Hohenauer, M., Leupers, R., and Meyr, H.: 'Instruction scheduler generation for retargetable compilation', *IEEE Design Test Comput.*, 2003, **20**, (1), pp. 34–41
33 Siska, C.: 'A processor description language supporting retargetable multi-pipeline DSP program development tools'. Proc. Int. Symp. on System Synthesis (ISSS), 1998, pp. 31–36
34 Morimoto, T., Yamazaki, K., Nakamura, H., Boku, T., and Nakazawa, K.: 'Superscalar processor design with hardware description language aidl'. Proc. Asia Pacific Conf. on Hardware Description Languages (APCHDL), 1994
35 Sato, J., Alomary, A., Honma, Y., Nakata, T., Shiomi, A., Hikichi, N., and Imai, M.: 'Peas-i: a hardware/software codesign systems for ASIP development', *IEICE Trans. Fundam.*, 1994, **77-A**, (3), pp. 483–491
36 Inoue, A., Tomiyama, H., Eko, F., Kanbara, H., and Yasuura, H.: 'A programming language for processor based embedded systems'. Proc. Asia Pacific Conf. on Hardware Description Languages (APCHDL), 1998, pp. 89–94
37 Rajesh, V., and Moona, R.: 'Processor modelling for hardware software codesign'. Proc. Int. Conf. on VLSI Design, 1999, pp. 132–137
38 ARC Cores. http://www.arccores.com
39 http://www.axysdesign.com. Axys Design Automation
40 Tensilica Inc: http://www.tensilica.com
41 Marwedel, P., and Goossens, G.: 'Code generation for embedded processors' (Kluwer Academic Publishers, 1995)
42 Aho, A., Sethi, R., and Ullman, J.: 'Compilers: principles, techniques and tools' (Addition-Wesley, 1986)
43 Hohenauer, M., Scharwaechter, H., Karuri, K., Wahlen, O., Kogel, T., Leupers, R., Ascheid, G., Meyr, H., Braun, G., and Someren, H.: 'A methodology and tool suite for c compiler generation from adl processor models'. Proc. Design Automation and Test in Europe (DATE), 2004, pp. 1276–1283
44 Zhu, J., and Gajski, D.: 'A retargetable, ultra-fast, instruction set simulator'. Proc. Design Automation and Test in Europe (DATE), 1999
45 Cmelik, R., and Keppel, D.: 'Shade: A fast instruction set simulator for execution profiling', *ACM SIGMETRICS Perform. Eval. Rev.*, 1994, **22**, (1), pp. 128–137
46 Witchel, E., and Rosenblum, M.: 'Embra: Fast and flexible machine simulation', in 'Measurement and modelling of computer systems' (1996), pp. 68–79
47 Nohl, A., Braun, G., Schliebusch, O., Leupers, R., Meyr, H., and Hoffmann, A.: 'A universal technique for fast and flexible instruction set architecture simulation'. Proc. Design Automation Conf. (DAC), 2002, pp. 22–27
48 Reshadi, M., Mishra, P., and Dutt, N.: 'Instruction set compiled simulation: A technique for fast and flexible instruction set simulation'. Proc. Design Automation Conf. (DAC), 2003, pp. 758–763
49 Improv Inc. http://www.improvsys.com
50 Itoh, M., Higaki, S., Takeuchi, Y., Kitajima, A., Imai, M., Sato, J., and Shiomi, A.: 'Peas-iii: an ASIP design environment'. Proc. Int. Conf. on Computer Design (ICCD), 2000
51 Hadjiyiannis, G., Russo, P., and Devadas, S.: 'A methodology for accurate performance evaluation in architecture exploration'. Proc. Design Automation Conf. (DAC), 1999, pp. 927–932
52 Itoh, M., Takeuchi, Y., Imai, M., and Shiomi, A.: 'Synthesizable HDL generation for pipelined processors from a micro-operation description', *IEICE Trans. Fundam.*, 2000, **00-A**, (3)

53 Schliebusch, O., Chattopadhyay, A., Steinert, M., Braun, G., Nohl, A., Leupers, R., Ascheid, G., and Meyr, H.: 'RTL processor synthesis for architecture exploration and implementation'. Proc. Design Automation and Test in Europe (DATE), 2004, pp. 156–160
54 Schliebusch, O., Hoffmann, A., Nohl, A., Braun, G., and Meyr, H.: 'Architecture implementation using the machine description language LISA'. Proc. Asia South Pacific Design Automation Conf. (ASPDAC)/Int. Conf. on VLSI Design, 2002, pp. 239–244
55 Mishra, P., Kejariwal, A., and Dutt, N.: 'Rapid exploration of pipelined processors through automatic generation of synthesizable RTL models'. Proc. Rapid System Prototyping (RSP), 2003, pp. 226–232
56 Mishra, P., Kejariwal, A., and Dutt, N.: 'Synthesis-driven exploration of pipelined embedded processors'. Proc. Int. Conf. on VLSI Design, 2004
57 Mishra, P.: 'Specification-driven validation of programmable embedded systems'. PhD thesis, University of California Irvine, March 2004
58 Mishra, P., and Dutt, N.: 'Automatic modelling and validation of pipeline specifications', *ACM Trans. Embedded Comput. Syst.*, 2004, **3**, (1), pp. 114–139
59 Mishra, P., Dutt, N., and Nicolau, A.: 'Automatic validation of pipeline specifications'. Proc. High Level Design Validation and Test (HLDVT), 2001, pp. 9–13
60 Mishra, P., Tomiyama, H., Halambi, A., Grun, P., Dutt, N., and Nicolau, A.: 'Automatic modelling and validation of pipeline specifications driven by an architecture description language'. Proc. Asia South Pacific Design Automation Conf. (ASPDAC) / Int. Conf. on VLSI Design, 2002, pp. 458–463
61 Mishra, P., and Dutt, N.: 'Modeling and verification of pipelined embedded processors in the presence of hazards and exceptions'. Proc. Distributed and Parallel Embedded Systems (DIPES), 2002, pp. 81–90
62 Mishra, P., Dutt, N., and Tomiyama, H.: 'Towards automatic validation of dynamic behaviour in pipelined processor specifications', *Des. Autom. Embedded Syst.*, 2003, **8**, (2-3), pp. 249–265
63 Mishra, P., Tomiyama, H., Dutt, N., and Nicolau, A.: 'Automatic verification of in-order execution in microprocessors with fragmented pipelines and multicycle functional units'. Proc. Design Automation and Test in Europe (DATE), 2002, pp. 36–43
64 Mishra, P., and Dutt, N.: 'Graph-based functional test program generation for pipelined processors'. Proc. Design Automation and Test in Europe (DATE), 2004, pp. 182–187
65 Mishra, P., and Dutt, N.: 'Automatic functional test program generation for pipelined processors using model checking'. Proc. High Level Design Validation and Test (HLDVT), 2002, pp. 99–103
66 Mishra, P., Dutt, N., Krishnamurthy, N., and Abadir, M.: 'A top-down methodology for validation of microprocessors', *IEEE Des. Test Comput.*, 2004, **21**, (2), pp. 122–131
67 Hoffmann, A., Schliebusch, O., Nohl, A., Braun, G., Wahlen, O., and Meyr, H.: 'A methodology for the design of application specific instruction set processors (asip) using the machine description language LISA'. Proc. Int. Conf. on Computer-Aided Design (ICCAD), 2001, pp. 625–630
68 Nohl, A., Greive, V., Braun, G., Hoffmann, A., Leupers, R., Schliebusch, O., and Meyr, H.: 'Instruction encoding synthesis for architecture exploration using hierarchical processor models'. Proc. Design Automation Conf. (DAC), 2003, pp. 262–267
69 Schliebusch, O., Kammler, D., Chattopadhyay, A., Leupers, R., Ascheid, G., and Meyr, H.: 'Automatic generation of JTAG interface and debug mechanism for ASIPs'. GSPx, 2004
70 Biswas, P., and Dutt, N.: 'Reducing code size for heterogeneous-connectivity-based VLIW DSPs through synthesis of instruction set extensions'. Proc. Compilers, Architectures, Synthesis for Embedded Systems (CASES), 2003, pp. 104–112
71 Mishra, P., and Dutt, N.: 'Functional coverage driven test generation for validation of pipelined processors'. Proc. Design Automation and Test in Europe (DATE), 2005
72 Mishra, P., Dutt, N., and Nicolau, A.: 'Functional abstraction driven design space exploration of heterogeneous programmable architectures'. Proc. Int. Symp. on System Synthesis (ISSS), 2001, pp. 256–261
73 Braun, G., Nohl, A., Sheng, W., Ceng, J., Hohenauer, M., Scharwaechter, H., Leupers, R., and Meyr, H.: 'A novel approach for flexible and consistent 2-driven ASIP design'. Proc. Design Automation Conf. (DAC), 2004, pp. 717–722