Chapter 12

Synthesis

From Programming Level to Hardware and Software Implementations using a Unified Database driven Synthesis Framework

epubli, ISBN 9783746752228 (2018)

This Chapter addresses the challenges of the high-level synthesis of *AAPL* agents and the various agent processing platforms from programming level.

12.1 The Big Picture: All together

Designing and implementing MAS for multiple significantly different platforms deployed in heterogeneous network environments is still a superior challenge. A unified High-level synthesis framework should enable the design and simulation of MAS for such a heterogeneous processing environment including and most important hardware SoC designs using application-specific digital logic meeting the goal of miniaturization and material-integration.

An important key feature is given by a common programming model for the different platform approaches, effecting both the programming and synthesis models. The principle unified synthesis flow of agent processing platforms is shown in Figure *12.1*, covering application-specific and application-independent (programmable) platform approaches. Application-specific platforms offer the lowest resource requirements, whereas application-independent (programmable) platforms offers the highest flexibility.

There is one common agent programming language *AAPL* model but different processing architectures that must be covered by the synthesis flow creating stand-alone parallel hardware implementations, alternatively stand-alone software implementations, and behavioural simulation models, enabling the design and test of large-scale heterogeneous systems. The central parts of the synthesis framework is the Agent Behaviour and Agent Platform compiler, discussed in Section *12.16*, summarized and shown in Figure *12.3*.

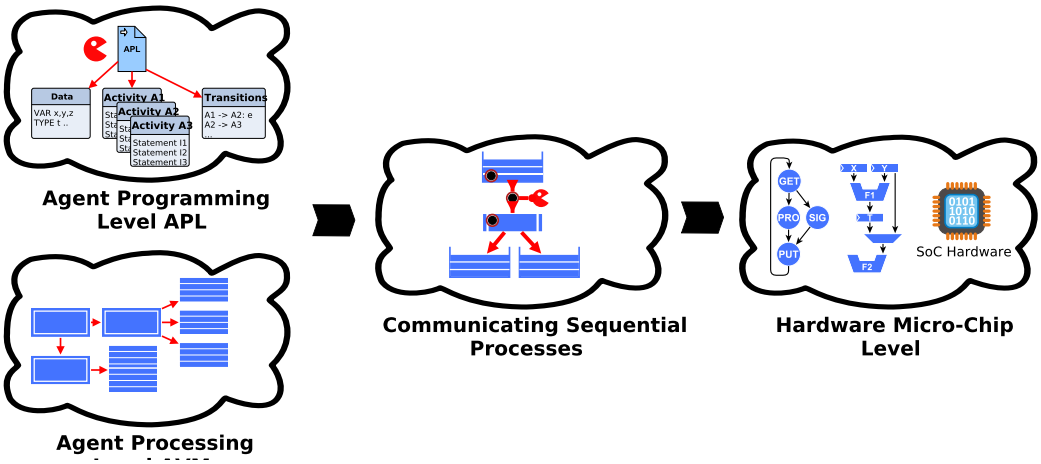

Fig. 12.1 *Road-map: The Simplified top-down level model hierarchy reflecting the synthesis flow for the hardware implementation of agent behaviour and processing.*

Application-specific agent synthesis embeds the agent behaviour in the platform, therefore the *AAPL* MAS programming model is entirely synthesized to platform building blocks, whereas the application-independent agent synthesis flow creates the platform (virtual machine) and the agent behaviour unit (program) separately, similar to a traditional hardware-software co-design.

The Agent compilers generate software models (*C*, ML), simulation models suitable for the *SeSAm* MAS simulator, and a high-level hardware model using the intermediate CCSP-based *ConPro* programming model, synthesized by the *ConPro* compiler to hardware behaviour model, discussed in Section 12.5, which can be finally synthesized to gate-level digital logic using common FPGA and ASIC synthesis tool chains.

There is a synthesis development kit that supports and eases the development of large multi-compiler frameworks and glues most software components part of the Agent synthesis framework with a unified database approach, discussed in the next section, and summarized in Figure 12.2. A graph-based virtual database driven hardware and software synthesis approach (*VDB*) should overcome limitations in traditional compiler designs.

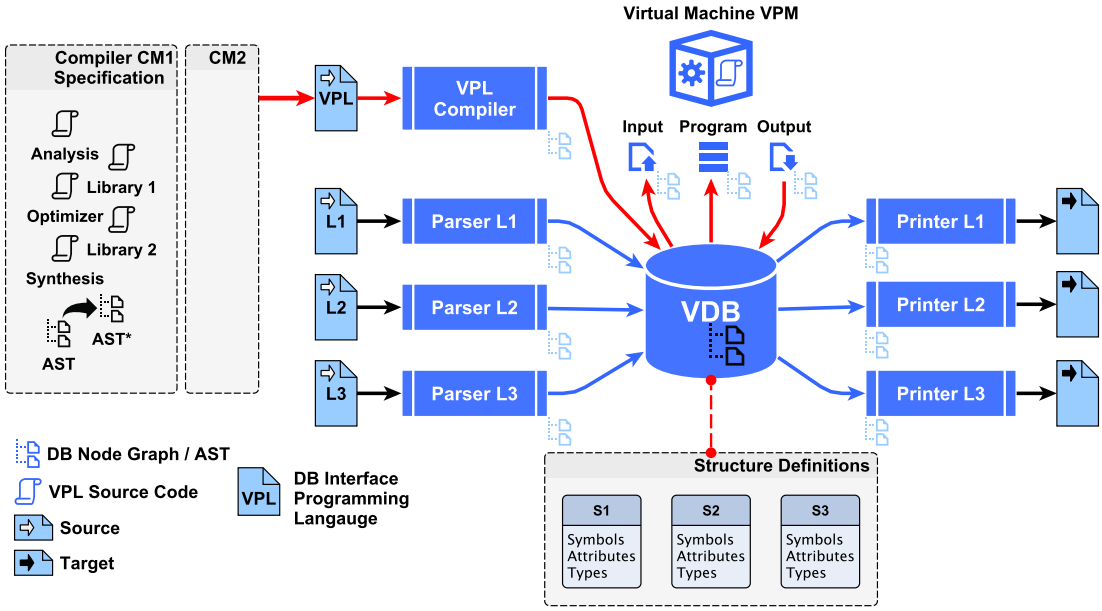

Fig. 12.2 *System architecture of the Virtual Database (VDB) driven Synthesis Development Kit SynDK implementing parser and formatted printer for a set of languages L, and a set of compilers C performing operations on abstract syntax trees AST (analysis, optimization, synthesis).*

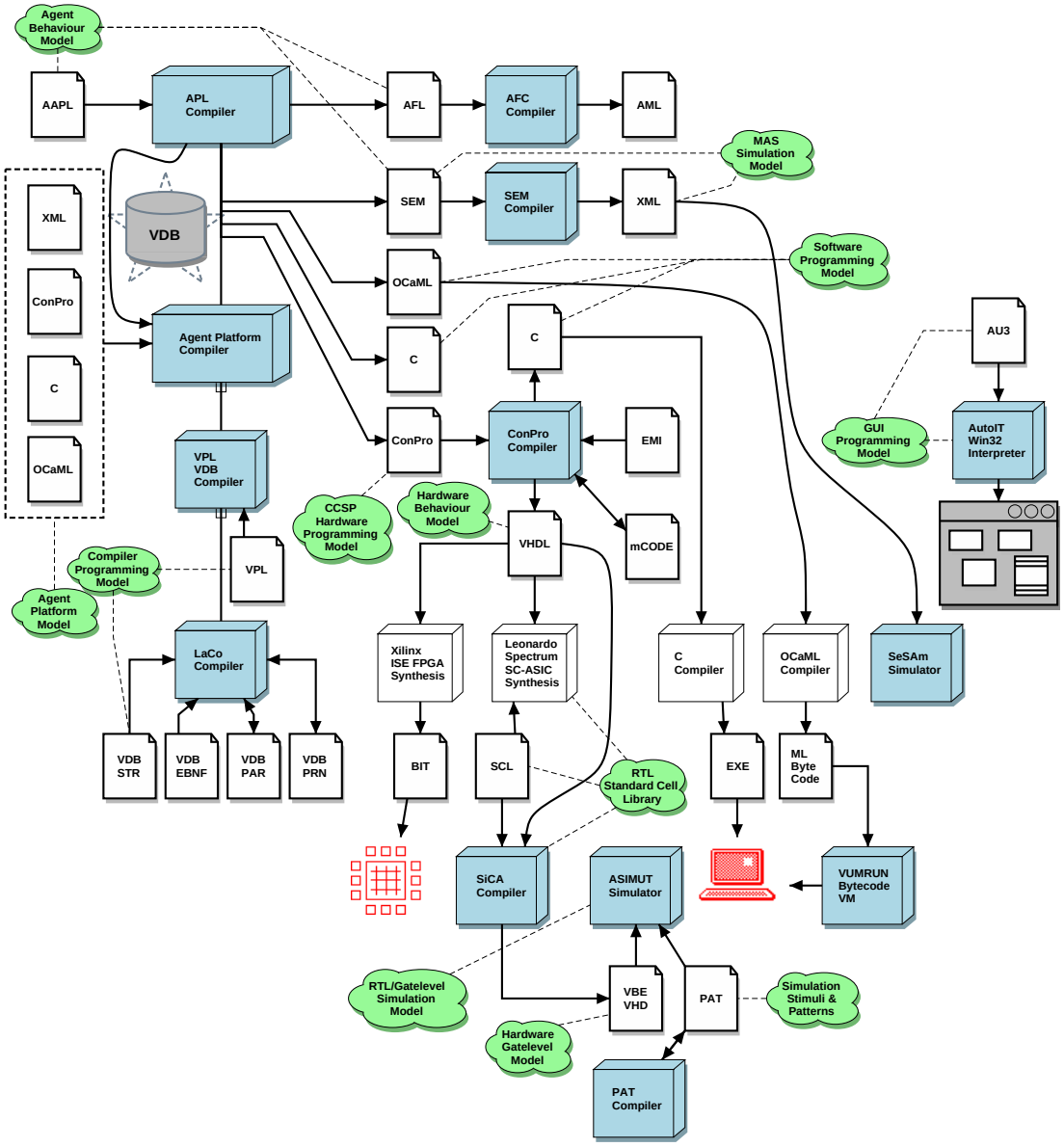

Fig. 12.3 *The Agent Synthesis Framework: covers the synthesis from Agent Behaviour to Hardware, Software, and Simulation Platform levels.*

It enables common synthesis from a set of source programming models and languages *IPL* to a set of destination models and languages *OPL* like hardware behaviour models with parsers translating text to graph structured database content and printers creating text from data base content, shown in Figure *12.2*.

12.2 SynDK: The Synthesis Development Toolkit

12.2.1 The Graph Database centric Synthesis Approach

Large CAD programs like compiler and synthesis tools traditionally are built from a large set of different data structures and types (by using composition of product and sum types). They are usually implemented by partitioning the program in different modules and components interacting with each other by using data structures or files, too, providing some degree of coupling and creating mainly a monolithic processing system. Very complex synthesis tools (e.g. digital logic gate-level or System-on-Chip high-level synthesis) are additionally partitioned in different independent programs interchanging data by using files with a well-known structure. Compiling of software or synthesis of hardware means the transformation of a set of input languages to a set of output languages usually performed in a pipelined data flow (different analysis, verification, optimization, and synthesis passes) with different internal list- or graph-like data structure representations. One example is the symbol table and linked references of symbols in statements.

A graph database driven compiler and high-level synthesis tool kit *SynDK* offering advanced operational resource sharing for a large set of input and output languages was developed to provide a unified synthesis framework and syntax-directed synthesis. The database approach is suitable for managing information with inherent graph-like nature and improves and ease the test and debugging of new compilers and ease the addition of new language front-ends and platform back-ends, and was a prerequisite to master the complex agent-on-chip synthesis flow in this work. A database model should address the structuring and description of the data, its maintainability and the form to retrieve or query the data [ANG08]. This concludes that a database-model is defined as a combination of three components:

1. A collection of data structures and structure types;
2. A collection of operators accessing the structures;
3. A collection of general integrity rules ensuring the satisfaction of structure relationships and rules (schemas).

Using a database in a compiler offers relationships between input data (parsed abstract syntax trees of a specific programming language), output data (synthesis results), and internal data structures (product and sum types, symbol tables) used by the compiler or synthesis tools. The data relationships and the data content are stored in the database and structured by the database-model.

In situations where information about the inter-connectivity or the topology of the data is more important, or as important as, the data itself, graph data-

base-models are preferred. Introducing graphs as a modelling tool has several advantages for this type of data [ANG08].

12.2.2 Virtual Database Organization

The Virtual Database (*VDB*) is organized by elements, nodes of a graph structure, represented by internal nodes (i-nodes), shown in Figure *12.4*. I-nodes represent elements of data records, and an i-node graph of linked i-nodes represent data structures. A database element can have attributes and is connected with other elements by linking them with other database elements. A node element has a unique type and optionally a name or value (type-tagged value). The links are mainly contained in the content table of an element. Though a database element (and the i-node) is generic itself, the content and the arrangement of elements in a graph like ordering (the structure) is constrained by types and structure relations given by a Structure Type Definition (*STD*), defining the types of database elements (nodes, type kind TYPE), the allowed element content for a specific node type (two-column row table providing links to child elements) and row lists attaching attributes to elements (type kind ATTR), and the order they may appear.

Attribute and content tables of an element are organized in rows, each row consisting of two columns each storing a tagged value, explained below.

There are generic elements called directories that have no specific structure, which can link to arbitrary database content (e.g. sub-graphs, like directories in file systems).

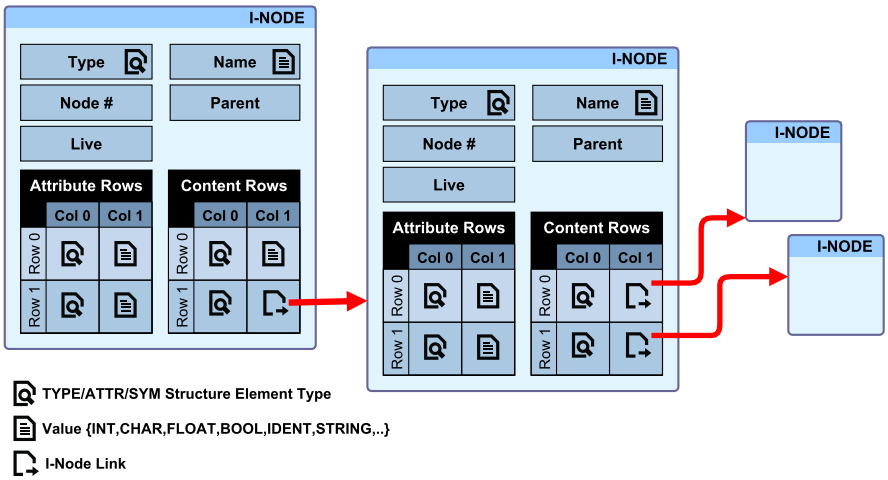

Fig. 12.4 *Database i-node body fields and connections of i-nodes using node reference links creating graph structures*

The element node (kind TYPE), attribute (kind ATTR), and symbol (terminal, kind SYM) types are internally represented and classified by a unique tagged numeric tuple constructor:

```
Type (SID,TID) with SID,TID ∈ ℕ
Attr (SID,TID)
Sym  (SID,TID)
```

where *SID* is a unique structure type definition identifier, and *TID* a type class (TYPE/ATTR/SYM) and structure specific unique type enumerator.

New type identifiers can be added at run-time. For example, a simple expression structure can be composed of elements, attributes, and symbols of the following structure types:

```
TYPE={expr=Type(EXP,1), value=Type(EXP,2), variable=Type(EXP,3)}
ATTR={operator=Attr(EXP,1)}
SYM={add=Sym(EXP,1), sub=Sym(EXP,2)}
```

These structure types have specific relations that are specified by the *STD*, discussed later.

A row of a content table of an element (the node) consists of two columns, usually specifying the type of a linked child node and the node link itself. A row of an attribute list usually consists of the attribute type and the value, which can be composed of a sub-structure encapsulated by a row list. Therefore, attributes of database elements can have arbitrary nested tree structures, too.

Regular expressions specify the possible element (node) child element types, their order, and the possible attributes for each element type. The child and attribute structure can be specified with repeating lists, (ordered or unordered) conjunction sub-structures (product types), and disjunction lists (sum types).

The VDB content can be mapped on, exported to, and imported from *XML* files, and the Structure Type Definition schemas discussed below can be mapped on *DTD* schema specifications.

12.2.3 The VDB Software Architecture

The graph-based database is the central part in the design and construction of compilers, shown in Figure *12.5*. Around the database there is a two-layered compiler architecture, consisting of native code compiler modules programmed with the ML programming language, for example, parsers, directly operating on the database, and interpreted compiler modules using the VDB programming language *VPL*, discussed in Section *12.2.8*.

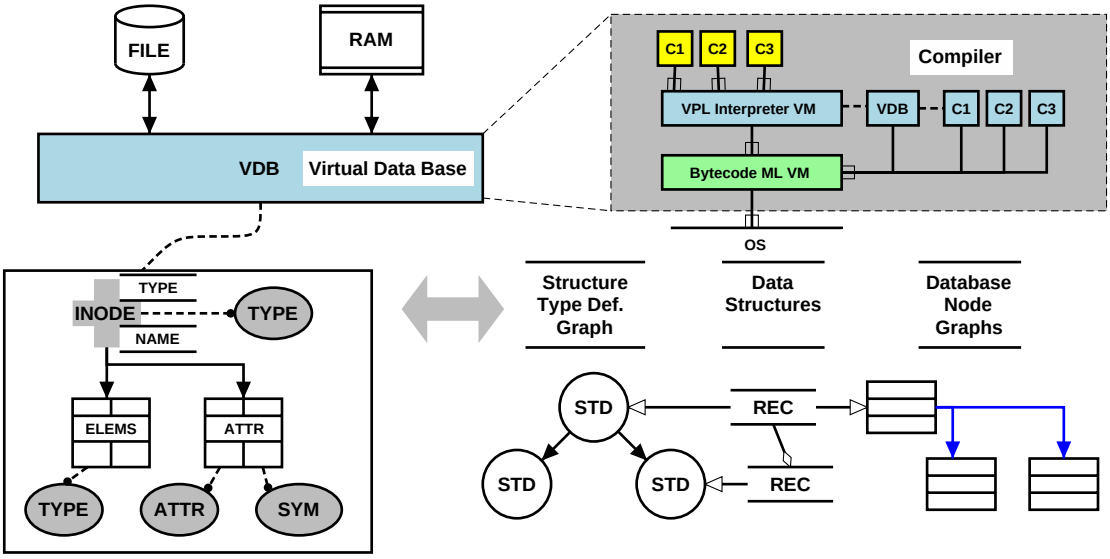

Fig. 12.5 *VDB design architecture overview and the relationship of data structures to i-node graphs.*

The advantages of this two layered compiler architecture approach are:

1. The high flexibility, interpreted versa compiled;
2. Multiple different programming languages can be supported by one synthesis program. All compilers and all compiler modules can exchange data through the database;
3. A *VDB* interface programming language (*VPL*) optimally matching the database and compiler programming that eases the design, and native code (ML) for optimized speed.

12.2.4 Structure Type Definition Schemas

A structure schema is defined by using the *S* language specifying the structure type definition used in the virtual database system for content verification at run-time, for the database programming interface, and for the construction of parsers and printers. A structure schema defines the relationship of database elements, and maps data structures on database content. Structure definitions are hierarchical.

There are core *STD* modules used and shared by all compilers and programming languages (or at least a subset of them), shown below. *STD* schemas can be cross referenced offering forward and backward references. For example,

the *SYM* symbol table schema includes expressions, and the *EXP* expression schema references symbols from the *SYM* schema.

GEN	Definition of generic attributes and elements like source code position information attributes and transformation elements that can added to any other element
TYP	Definition of basic data types
EXP	Definition of expressions
SYM	Definition of symbol tables (providing symbols for composed data types, storage object definitions, classes, function interfaces, and many more)
INSTR	Definition of imperative and functional statements (using the expression module)

A structure type definition schema (see Definition *12.1*) is partitioned in type, attribute, and symbol definitions, shown below. The *STD* uses structural regular expression to specify the structure of the type elements. Structural regular expressions define a list of structure elements, which can be regular expressions, too. There are ordered and unordered structures. In ordered structures, the elements must appear in the order they were specified. Structural regular expression are used to specify the attribute signature of database elements (types), too.

Def. 12.1 *Structure Type Definition Language S using structural regular expressions defining the element graph structure and the sub-structure with attributes*

```
STRUCTURE Descriptive name -> STR identifier;
DESCRIPTION "text";
TYPES
BEGIN
   type-name [attributes] := regular-structure-expression;
   ..
END;
ATTRIBUTES
BEGIN
   attr-name := regular-structure-expression;
   ..
END;
SYMBOLS                    Regular Expressions:
BEGIN                      a,b,c,...    Ordered Structure
   sym-name;               a&b&c&...    Unordered Structure
   ..                      (a|b|c|..)   Choice
END;                       (..)+ (..)*  Lists of Structure
```

A *STD* is a super-class of the commonly used *DTD* schema specification or XML documents, primarily due to the lack of attribute tree structures. But a set of *STD*s can be transformed to a *DTD*, expanding nested element attribute structures to XML elements.

A *STD* schema can be extended (but not be changed) at run-time creating a modified *STD* by adding new symbols, attributes, and types, preserving the initial (or previous) *STD* as a subset without violating the previous usage of *STD*s. This feature offers a kind of dynamic typing system at run-time. Additional *STD*s can be added at run-time, too, which can be used immediately after their definition.

A simple structure schema definition is shown in Example *12.1*.

Ex. 12.1 *A simple Structure Type Definition (STD Schema) for expressions*

```
STRUCTURE Expression -> EXP;
DESCRIPTION "Expression Structure";
TYPES
BEGIN
   element [selector? & TYP.datasubtype] := IDENT;
   expr [operator & guarded?] := (element | expr | value)+;
  value [format & time? & TYP.datasubtype] := INT|FLOAT|BOOL|STRING|CHAR;
END;
ATTRIBUTES
BEGIN
   a := INT|element|expr;
   b := INT|element|expr;
   direction := up|down;
   index := (INT|element|expr)+;
   method := IDENT;
   operator := add|sub|mul|div;
   range := a,b,direction;
   selector := (range|index|method|struct)+;
   size := INT;
   struct := IDENT;
   time := INT,(ps|ns|us|ms|sec)?;
   format := (%decimal|%float|%string|%char|%bool);
END;
SYMBOLS
BEGIN
   add;
   %bool -> FmBoolean;
   %char -> FmChar;
   %decimal -> FmDecimal;
   div;
   down;
   %float -> FmFloat;
   %hex -> FmHex;
   ...
END;
```

12.2.5 Tagged Values

A tagged value is a sum type, shown in Definition *12.2*, which encapsulates different data values like integers, boolean, strings, but also node links, identifiers, symbol, attribute, and type element identifiers (and many more) by assigning a data type tag to the value. This tagged value is a fundamental concept of the database design and the design of parsers mapping grammar to database content, formatted printers, and the database programming language *VPL* explained later.

Def. 12.2 *Tagged value sum type definition and the corresponding tagged type signature*

```
TYPE tagged_value =
   | Int (int value)
   | Int64 (int64 value)
   | Float (float value)
   | Char (char value)
   | Bool (boolean value)
   | String (string value)
   | Node (node reference)
   | Row (row value)
   | Rows (row list)
   | Table (row array)
   | Ident (string value)
   | Type (S.id)
   | Attr (S.id)
   | Sym (S.id)
   | Lab (S.id)
   | Typid (tagged_type)

TYPE tagged_type =
     | INT
     | INT64
     | FLOAT
     | CHAR
     | BOOL
     | STRING
     | NODE
     | ROW
     | ROWS
     | TABLE
     | IDENT
     | TYPE
     | ATTR
     | SYM
     | LAB
     | TYPID

TYPE row = (tagged_value,tagged_value)
```

12.2.6 Programming of Compilers and LaCo: The Language Compiler

An enhanced and modified *OCaML* programming language (based on *OCaML* 3, details in [REM02]) and byte-code interpreter VM (*VUM*-ML, based on [OCA03]) is used to implement the VDB and related program libraries. One major extension of *OCaML* is the addition of symbolic enumeration types required for the definition of type descriptors *Type(SID,TID)*, *Attr(SID,TID)*, and *Sym(SID,TID)* in ML. *OCaML* provides functional, imperative, and object orientated programming models in one consistent programming language. The ML source code can be compiled to native machine or byte code executed by a VM. The byte code approach is used throughout this entire work to ensure portability and flexibility.

New compilers can either be programmed in *OCaML* or by using the interpreted *VPL* database programming language, introduced in Section *12.2.8*. A hybrid approach is commonly applied involving ML and *VPL* code, too. At least the parser are included in the ML program itself. Programming of parsers

usually requires a large amount of time. *VDB* parsers map a specific programming language directly to VDB elements and database content. The structure of the database content is specified by a *STD* schema. Common language constructs like expressions are reflected by the core schema definitions.

To simplify the generation of parsers (including lexers) and formatted printers, the LaCo compiler was created.

It can map a *STD* schema with an extended Backhaus-Nauer Form (*EBNF*) syntax specification to parsers and printers automatically, shown in Figure *12.6* on the right side. A compiler and synthesis framework program supports the processing of multiple languages $L=\{L_1, L_2, ..\}$, which can be input, output, input and output, or intermediate languages.

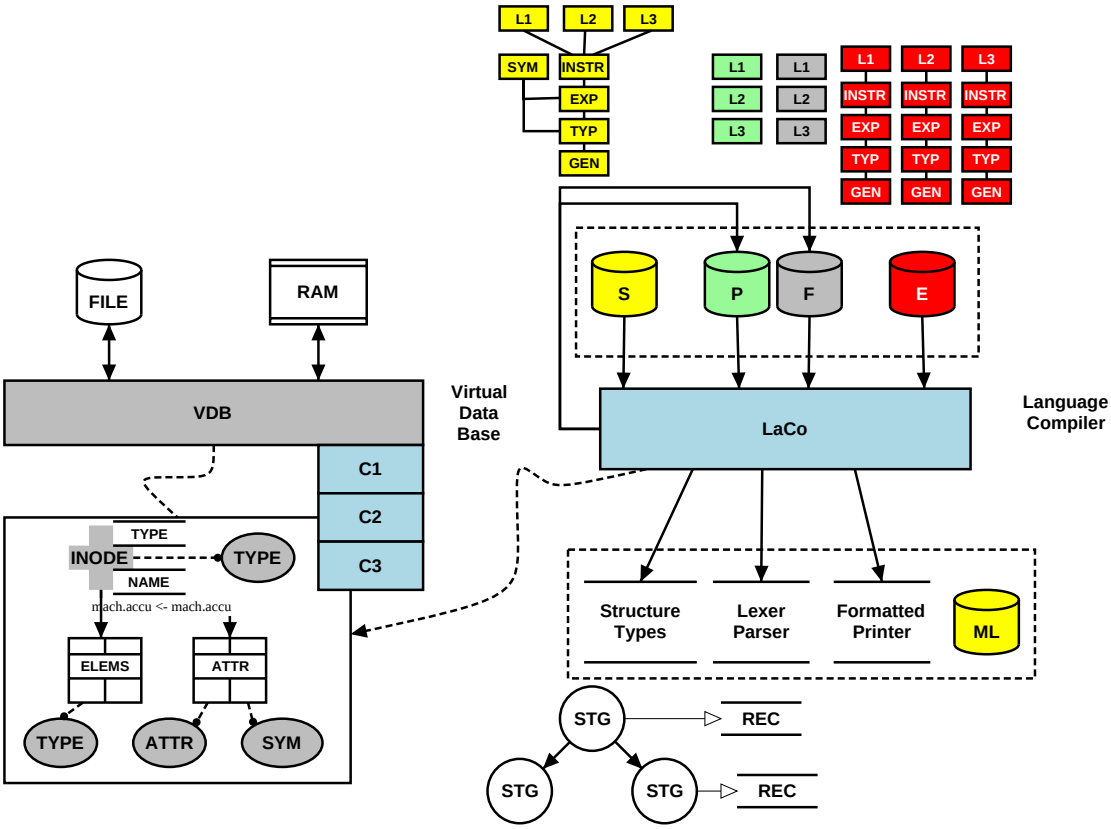

Fig. 12.6 *The Language Compiler LaCo (middle), the structure definition graphs (STD, top) and the relationship with the database (left).*

The *LaCo* compiler creates from a *STD/EBNF* definition set {S_1, S_2, .. , E_1, E_2, ..} one parser definition *P* and one formatted printer definition *F* for each input and output programming language to be processed by a compiler. Alternatively, the *P* and *F* definitions can be programmed directly by the user. Finally, *LaCo* compiles the *P* and *F* definitions to ML source code, which can be integrated in a *VDB*-based compiler program providing access to the *VPL* part of the compiler.

The *EBNF* specification with the *E* language follows the same structure existing in the *S* language. There are three main sections in an *E* specification module: type element syntax, attribute syntax, and symbol syntax rules, again using regular expressions.

Def. 12.3 *EBNF Syntax Specification Language E*

```
SYNTAX Descriptive name;
STRUCTURE STR-NAME;
DESCRIPTION "text";
TYPES
BEGIN
  type-name ::= regular-expression .
  ..
END;
ATTRIBUTES
BEGIN
  attr-name ::= regular-expression .
  ..
END;
SYMBOLS
BEGIN
  sym-name ::= regular-expression .
  ..
END;
⟦ EVALUATE
  BEGIN
    eval-rules+
  END; ⟧
```

12.2.7 Abstract Syntax and VDB Graphs

A parser for a specific language, generated from an *EBNF* specification and *STD* schemas, parses source text and generates an Abstract Syntax Graph (AST) directly stored in the database as a database element node graph. This approach enables syntax-driven synthesis and unifies the parser design significantly.

epubli, ISBN 9783746752228 (2018)

$x.[i]+(y-f(a,n-1))$

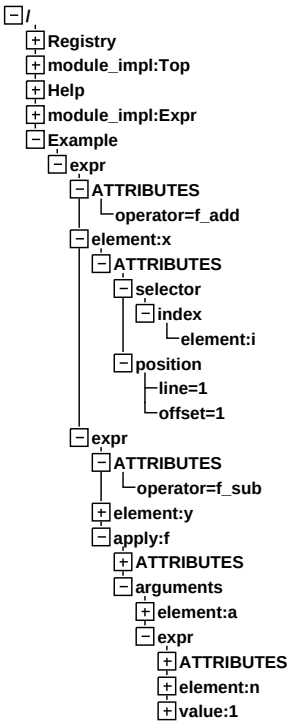

Fig. 12.7 *Example of an AST parsed from syntax "x.[i]+(y-f(a,n-1))" and stored in the database (ATTRIBUTES belong to the parent database element)*

This syntax-driven synthesis enables the composition of complex synthesis frameworks from a large set of different compilers, which can exchange data though the well-structured ASTs.

Figure 12.7 shows the *VDB* representation of the parsed AST derived from a simple expression.

12.2.8 VPL: VDB Programming Interface and Interpreter

VPL is an imperative high-level programming language derived from *Modula* providing direct access to the virtual database *VDB*. Beside common imperative programming statements like assignments, branches, loops, and functions there is advanced support for database content query and modification operations including node constructors and paths. *VPL* offers direct access to structure of the content of the database, e.g., i-node graphs; by path selectors for query and constructors insert operations.

Figure 12.8 shows the syntax-directed *VPL* compiler and run-time system. A *VPL* source program text is parsed and stored as an AST in the database. A *VPL* compiler performs semantic checking and analysis of this AST, finally linking all functions, objects, storage objects and external library references.

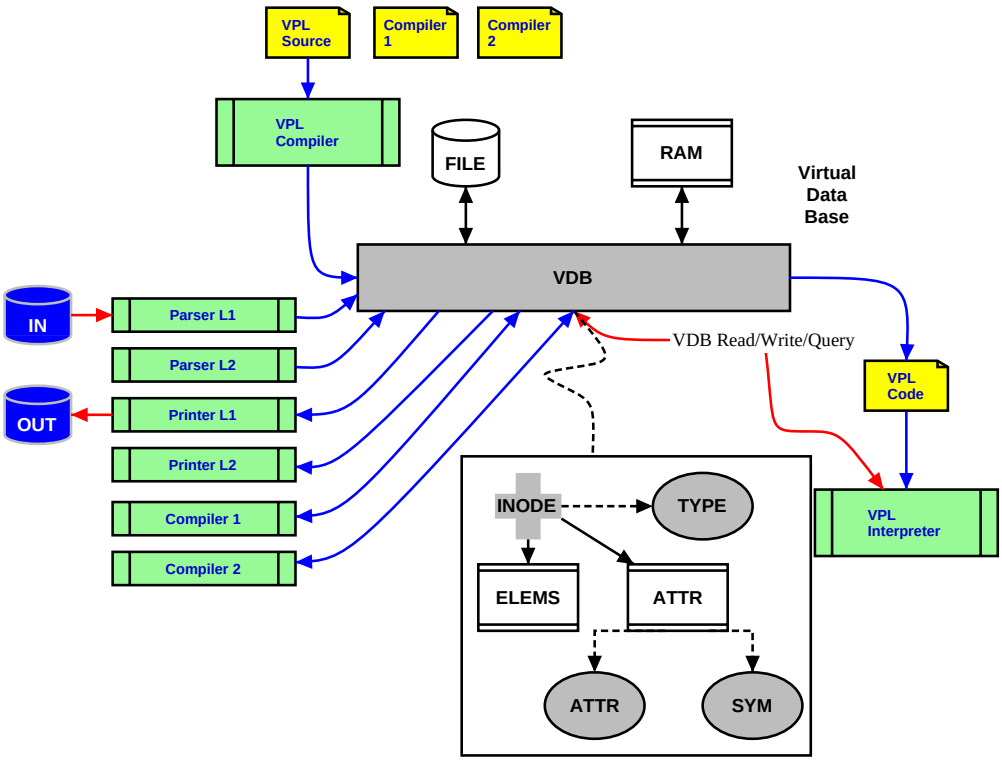

Fig. 12.8 *The VPL interpreter processes pre-compiled VPL programs directly from the database. VPL programs implement most parts of a compiler (excluding parser) and interacts directly with the DB.*

A *VPL* VM executes (interprets) this pre-compiled AST directly from the database. *VPL* can call built-in custom and common compiler and library functions (programmed in ML), and built-in ML functions can call *VPL* functions by invoking the *VPL* interpreter. A pre-compiled VPL program can be saved by dumping the database content to a file, which can be later loaded by the custom compiler to be implemented.

Data is handled in *VPL* by using tagged values. A tagged value is a tuple consisting of a type tag and a data value. The type of a tagged value can be extracted any time by using the `x->TYPE` path selector expression. These tagged values can be directly handled by the database VDB. Tagged values can be composed of rows (`Rows`) or value arrays of tagged values (`Table`). A row is a composed value of two tagged values, often used in database tables. Database elements and their attributes have specific element types {`Type`, `Attr`, `Sym`, `Lab`}, which can be handled by a tagged value and created by using constructors. *VPL* offers overloaded operations applied to tagged values, for

example, the addition operation can be used for integers, floats, strings, and lists (rows).

There are no user defined product types. Instead object orientated programming is used to implement record structures and operations that are applied to the data. Primarily, the graph based *VDB* is used to store and handle record structured data. There are simple sum types limited to constant enumerations.

Data processing in *VPL* is performed always with tagged data values. That means variables, object, and function parameters are polymorphic. Optionally types of variables and function, procedure, and method parameters can be restricted to a specific data type checked at run-time using type constraints applied to variable or parameter definitions. The run-time check can be disabled.

VPL Constructors

Constructors can be used to create rows, list of rows, element nodes, and search patterns required for query/lookup operations applied to the database content. The most commonly used constructor formats are summarized below.

`a:b`

Row constructor consisting of two values a and b composing the row tuple (*a*,*b*)

$[a_1:b_1;a_2:b_2;\ldots]$

Row or attribute list constructor

`S.t`

Type constructor (Element type, attribute, or symbol, depending on the respective structure definition)

`S.t v`

Attribute (row) constructor composing the row tuple (*S.t*,*v*) with a type constructor.

$S.t\ [a_1:b_1;a_2:b_2;\ldots]$

Alternative attribute (row) constructor composing the row tuple S.t:[a1:b1;a2:b2;...] assigning a row list (sub-structure) to the attribute.

`S.t [a][r]`
`S.t n [a][r]`

Element (node) constructor with optional element name *n*, attribute list [*a*], and element content table [*r*]. *S* is the structure identifier, *t* is

the (element) type identifier. The element name *n* can be either an identifier `Ident` "name", or any value, i.e., "string", 100.

`Value v`

Element `EXP.value v [] []` constructor. The value *v* can be of any type.

VPL Paths

Path selectors are always applied to variables referencing i-node content (or in some cases row lists). They apply filter patterns to the current content of the variable. Patterns can contain chained type, value, row and attribute selectors. The most important path selectors are summarized below.

$x \rightarrow p_1 \rightarrow p_2 \rightarrow ..$

Chained (nested) path selector. The selector elements p_1, p_2,.. are applied from left to right order.

$x \rightarrow S_1.t_1 \rightarrow S_2.t_2..$

A path selector with type constructors filtering element types, attributes, symbols, or labels (Structure S, type t).

`x→ATTR`
`x→ATTR→...`

Attribute table selector of i-nodes/database elements

`x→ROWS`
`x→ROWS→...`

Content table selector of i-nodes/database elements

`x→ROWS.[i]`
`x→ATTR.[i]`
`x→COLS.[i]`

Table row or column index selectors. First order `ROWS.[i]`, `ATTR.[i]`, and `COLS.[i]` selectors can be used on left-hand and right-hand side in assignments. *The first row and column index value is always 1.*

`x→NODE`
`x→NAME`
`x→VALUE`
`x→TYPE`
`x→NAME := ..`

Node selector (can be applied to a row or an i-node reference), name filter (can be applied to i-nodes), value filter (can be applied to a row

or an i-node), Type kind filter (can be applied to a row, value or i-node reference), and the modification of the i-node element name.

VPL Procedural Programming

The central programming paradigm of *VPL* is the composition of a compiler (or any other program) by a set of functions operating on database elements. User functions can be defined with a set of polymorphic typed formal parameters holding tagged values. The return type of a function is initially polymorphic, too. Function parameter and return types can be type constrained (checked at run-time). The function definition consists of the parameter interface, local variable definitions, and statements, shown below.

```
function f(p1,p2,...)
   var lV1,lV2,lV3,...;
   statements;
   return ε;
end;
x := f(a1,a2,..);
```

```
procedure p(p1,p2,...)
  var lV1,lV2,lV3,...;
  statements;
  return;
end;
p(a1,a2,..);
```

There are built-in functions and procedures providing commonly used operations, for example, list (rows) operations.

VPL Objects

Other than common programming languages like *Java* offers *VPL* support for object orientated programming in addition to the procedural programming model. Objects consists of private data (tagged value variables) and methods operating on the data. Individual objects are instantiated from an object class definition. The object class definition consists of an object class name, optional instantiation parameters, local variables representing the state of an object, and method definitions, modifying the state of an object. There exist no record structure types in *VPL*, therefore object classes are used instead to implement structures modified by their methods. Each object is extended with a set of methods required for object control only applicable to the object class type: new, fork, and destroy. The `new` method creates (instantiate) a new object from an object class type (with arguments if required). The `new` method will execute initialization instructions, if any. The `destroy` method deletes an object and executed finalization instructions, if any. The `fork` method creates a copy of the specified object copying the state of the object (content of variables), too. After a fork had happened, both objects are independent. The principle programming structure of an object class definition and the object instantiation is shown below.

epubli, ISBN 9783746752228 (2018)

```
object oc(x1,x2,...)
   var lV1,lV2,lV3,...;
   self o;
   method m(p1,p2,..) .. end;
   initialize .. end;
   finalize .. end;
end;
o := oc.new(e1,e2,..);
o.oc.m(a1,a2,..)
```

VPL Control and Data Processing Statements

VPL offers data processing with variables and expressions like any other imperative programming language. In contrast to commonly used programming languages most arithmetic and relational operations are polymorphic with respect to the supported tag types. For example, the addition operation can be used for integer, float, string, and list operands, partially with automatic type conversion if possible.

12.2.9 Compiling a Compiler

The construction of a synthesis framework for a set of compilers $CC=\{cc_1, cc_2, ..\}$ supporting a set of languages $L=\{l_1, l_2, ..\}$ invokes the following steps:

1. A specification of the top-level AST structure definition schema $S_{l,i}$ for each language l_i, inheriting the core structures *GEN*, *TYP*, *SYM*, *EXP*, *INSTR*.;
2. An *EBNF* syntax specification E_i for each programming language;
3. Generation of the parser and formatted printer specification (in intermediate *P/F* language) using *LaCO*, one P_i/F_i for each language;
4. Optionally, editing or adding *P/F* specifications manually (generation of formatted printer from *S/EBNF* specification not unique and not always complete);
5. Generation of ML parser, lexer, and printer for each supported language from the P/F specifications by using *LaCo*;
6. Programming of the built-in ML modules for each compiler, at least I/O wrappers for the parser/printer access from *VPL* level;
7. Compiling of the ML sources (parsers, printers, and built.-in compiler modules) producing the core *VDB* compiler (ML → byte code);

epubli, ISBN 9783746752228 (2018)

8. Programming of the *VPL* modules for each compiler and compiling the sources in the database;
9. Loading support data in the database required by the compilers, for example, templates, libraries, definition data;
10. Saving the database to a file (binary format);
11. Testing the compiler(s). Only the ML virtual machine, the byte code ML program, and the saved database is required, composing a multi-compiler synthesis framework.

12.2.10 Conditional and Parametric Compiling

Compiling, for example, of agent processing platforms, commonly invokes already modelled components, for example, a virtual machine or a tuple database, preferable given in the target programming language (*ConPro*, *C*, *XML*,..). Traditional pre-compiled and library based approaches are limited to static modules. But platform and application-specific synthesis requires parametrizable module blocks. To offer this synthesis feature, conditional and parametric compiling was introduced. This technique is known basically from the preprocessor used in several programming languages and compilers. But here the conditional and parametric compiler statements are parsed together with the source programming language and are processed by the compiler and synthesizer. They are available in the complete synthesis flow. Using the *VDB* approach these compiler statements can easily inserted in the AST of the source language by using generic wrapper statements defined in the *GEN STD* (generic element and attribute types may appear anywhere in the structure graphs, similar to source code positions and comment elements). Beneath statements there are compiler variables, part of the compiler environment, which can be used as parameters in the source programming language.

The conditional and parametric compiling approach allows the specification of code generators for various programming languages, used, for example, in the programmable Agent platform synthesis, discussed in Section *12.3.2*

An example for a *ConPro* module with parametric and conditional compile statements is shown below. A compiler statement starts with the # character (like conditionals processed by the C pre-processor), and a compiler parameter variable is prefixed with the $ character.

The ## operation concatenates compiler variables with identifier strings. Parameter variables can be provided externally by a parameter file or created by the compiler during the compilation. This approach overlays the program code with interpreted statements.

epubli, ISBN 9783746752228 (2018)

Ex. 12.2 *Parametric and conditional compiler statements inserted in a program code fragment*

```
#if $vm_static = true then
  process vm_static:
  begin
     array datasegment: reg [$ds_size] of integer[$data_width];
     ...
     #foreach $p in $processes do
       if $p##state = S_START then
       begin
         for i = 0 to $ds_size-1 do
         begin
           datasegment.[i] <- 0;
         end;
       end;
     #endfor
  end;
#else
  ...
#endif
```

12.3 Agent and Agent Platform Synthesis

Agent synthesis involves the compiling of agent run-time units (i.e., machine code and state container) from a behavioural programming model (here *AAPL*) and the design of the agent processing platform, commonly a virtual machine capable to process the agent machine code.

Two different approaches must be distinguished leading to two different synthesis flows:

1. Non-programmable and application specific agent processing platforms including the description of the agent behaviour using only one design flow;
2. Programmable generic agent processing platforms excluding the application-specific agent behaviour with two separate design flows for the platform (virtual machine) and the agent behaviour (machine code).

Furthermore, each approach provides hardware and software implementations (not to be confused with the agent software terminology in the second approach). Hardware and software implementations for each approach and platform class are compatible on the operational and communication level and can be deployed mixed in heterogeneous networks. The next two sections discusses both synthesis flows separately.

epubli, ISBN 9783746752228 (2018)

12.3.1 Non-programmable Agent Platform Synthesis

The AAPL model is a common source for the implementation of agent processing on hardware, software, and simulation processing platforms. The previously introduced database driven high-level synthesis approach is used to map the agent behaviour on these different platforms, shown in Figure *12.9*. The application-specific agent processing platform class implements the agent behaviour, that means the control machine based on the ATG model, and the data storage for a number of agents.

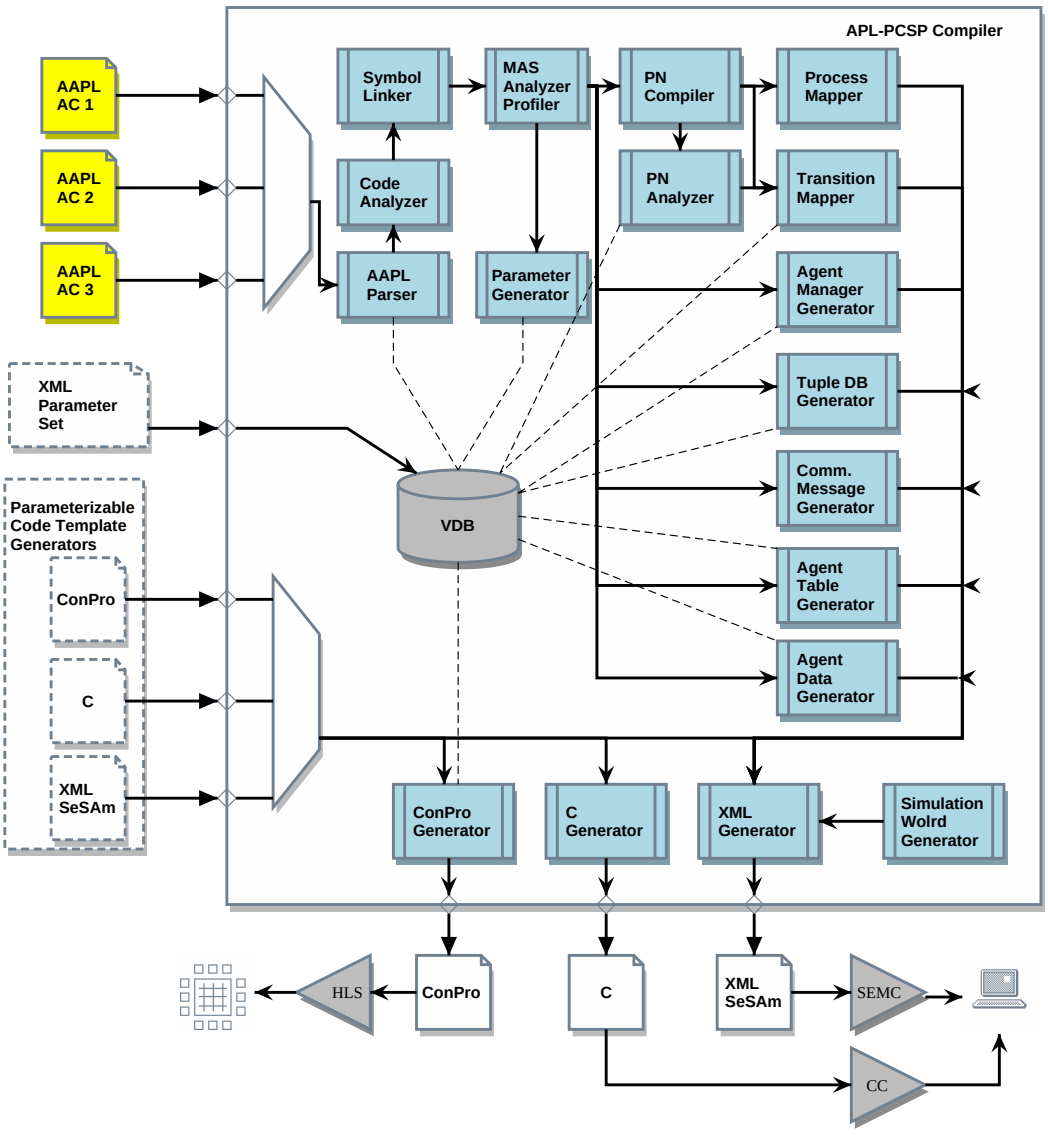

Fig. 12.9 *PCSP Compiler Flow*

Therefore, the agent processing platform required on each network node must implement processing units for different agent classes and must be scalable to the microchip level to enable material-integrated embedded system design, which represents a central design issue, further focussing on parallel agent processing and optimized resource sharing.

There are different internal compiler flows supporting the synthesis of hardware and software platforms. They do not differ significantly. The simulation model can be derived directly from the agent behaviour specification (the *AAPL* sources) using *SeSAm* agents, and enabling the behavioural simulation and testing of MAS and their algorithms. Alternatively, the *PCSP* agent processing platform itself is compiled in a simulation model (using *SeSAm* agents), enabling platform simulation and testing

PCSP Hardware Platform Synthesis

The HW platform design is application-specific with static resources, that means, all supported agent classes and the maximal number of agents processed at run-time must be fixed at design-time. All parts are integrated in one System-on-Chip (SoC) design on RT level.

A multi-stage High-level Synthesis approach is used to map the *AAPL* source code specification to micro-chip SoC level, shown in Figure *12.10*.

The *AAPL* sources are parsed and an AST is created that is stored in the compiler database. The first compiler stage analyses, checks, and optimizes the agent specification AST. The second stage is split basically in different parts: an activity to process-queue pair mapper with sub-state expansion, a transition network builder, manager generators, and a message generator supporting agent and signal migration.

This microchip-level processing platform implements the agent behaviour with the already introduced reconfigurable pipelined communicating processes architecture (*PCSP*). The activities and transitions of the *AAPL* programming model are merged in a first intermediate representation by using state-transition Petri Nets (PN), performed by a PN compiler, shown in Figures *12.9* and *12.10*.

This PN representation allows the following CSP derivation specifying the process and communication network, and advanced analysis like deadlock detection. Timed Petri-Nets can be used to calculate computational time bounds to support real-time processing, performed by the time analyser.

Keeping the PN representation in mind, the set of activities $\{A_1,..\}$ is mapped on a set of sequential processes $\{P_1,..\}$ executed concurrently, performed by the process mapper.

epubli, ISBN 9783746752228 (2018)

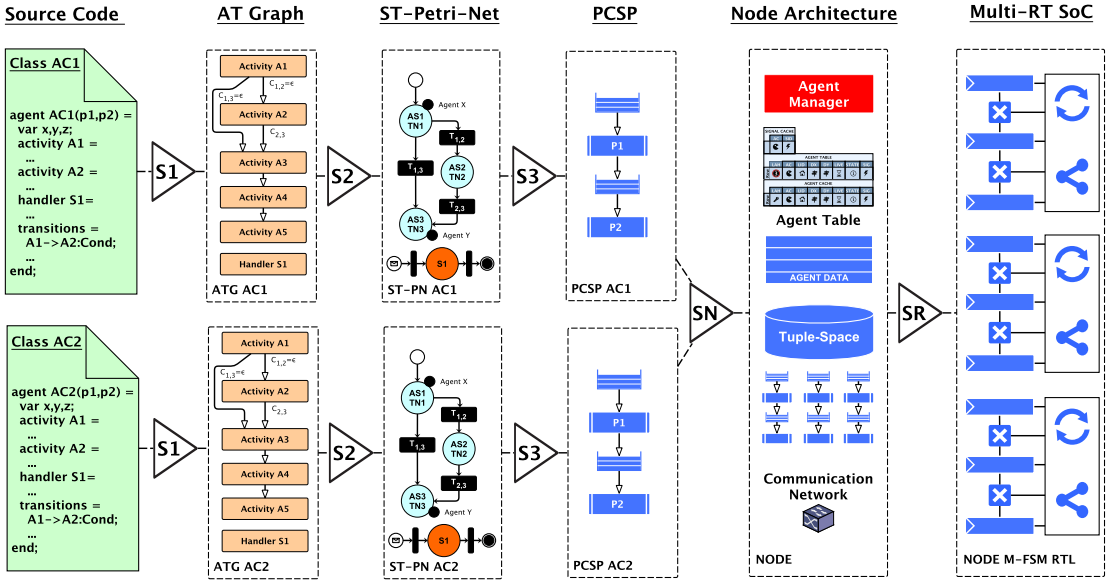

Fig. 12.10 *Synthesis Flow for the HW PCSP Agent Processing Platform*

Each subset of transitions $\{T_{i,j},..\}$ activating one common activity process P_j is mapped on a synchronous N:1 queue Q_j providing inter-activity-process communication, and the computational part for transitions embedded in all contributing processes $\{P_i,..\}$, performed by the transition network mapper. Changes (reconfiguration) of the transition network at run-time are supported by transition path selectors, handled by the transition mapper, too.

Each sequential process is mapped (by synthesis) on a finite-state machine and a data path using the Register-Transfer architecture (RTL) with mutual exclusive guarded access of shared objects, all implemented in hardware, outputting a *ConPro* programming model, finally processed by the *ConPro* compiler part.

The Agent Manager (AM) is responsible for the token injection and token passing of agents, which is generated by the Agent Manager mapper fed with information from the MAS and PN analyser.

To handle I/O-event and migration related blocking of statements, activity processes executing these statements are partitioned in sub-states A_i=>$\{a_{i,1}$, $a_{i,2}$, .. , $a_{i,TRANS}\}$ and a sub-state machine decomposing the process in computational, I/O statement, and transitional parts, which can be executed sequentially by back passing the agent token to the input queue of the process (sub-state loop iteration). This task is handled by the process and transition mappers, too. One sub-state partition example is shown below in Example *12.3*.

Ex. 12.3 *Sub-state and Sub-process Decomposition of an AAPL activity with blocking statements*

```
activity init =
  init1: dx := 0; dy := 0; h := 0; → init2
  init2: if dir <> ORIGIN then
           moveto(dir); ⊥ init3
           init3: case dir of
                    | NORTH => backdir:=SOUTH;
                    | SOUTH => backdir:=NORTH;
                    | WEST =>  backdir:=EAST;
                    | EAST =>  backdir:=WEST;
                  end; → init4
         else
           live:=MAXLIVE; backdir:=ORIGIN; → init4
         end;
  init4: group := Random(integer[0..1023]);
         out(H,id(SELF),0); → init5
  init5: rd(SENSORVALUE,s0?);  ⊥ initTRAN
  initTRAN: Transition Computation
```

Process replication can be applied to offer parallel processing of activity processes with high computation times to avoid data processing bottlenecks. The replication is performed by the process mapper based on the analysis results from the PN timing and MAS analyser.

PCSP Software Platform Synthesis

For performance reasons and the lack of fine-grained parallelism the ATG of an agent class is implemented differently:

- Each activity of an agent class and conditional expressions are implemented with ***functions***:

  $A_i \Rightarrow FA_i()\{I_1;I_2;..;\texttt{return;}\}$
  $T_{x,y}|_{Cond} \Rightarrow FC_{x,y}\{\texttt{return } cond;\}$

- Transitions are stored in dynamic lists:

```
struct tl {void (*from)();
           void (*to)();
           int (*cond)();
           struct tl *next;};
```

- Modification of the transitional network modifies the transition lists;

- Transitions between activities are handled by a ***transition scheduler***, which calls the appropriate activity functions:

```
while(!die) {next=schedule(curr,tl); curr=next; curr()}
```

- ***Multi-threading***: At run-time each agent is assigned to a separate thread executing the transition scheduler;
- Operating system or multi-threading primitives are used to synchronize agents (event, mutex, semaphore, timer);
- ***Creation*** of agents at run-time is performed by creating an agent handler thread;
- ***Migration*** of agents is performed by transferring the state of the agent (data state of all body variables and control state ➠ next activity after migration)

PCSP Simulation Platform

The *SeSAm* simulator environment [KLU09] is used to perform functional testing and analysis of multi-agent systems operating in distributed sensor networks.

- *SeSAm* models the agent behaviour with ATG, too. But: 1. The ATG is static at simulation-time; 2. Activities may not block; 3. There are no preemptive signal handlers.
- For this reason: a transition and signal scheduler handles the agent processing
- Activities with blocking statements (I/O) must be split according to the *PCSP* model
- Migration of agents is performed by changing the spatial location!

The simulation design flow includes an intermediate representation using the *SEM* programming language, providing a textual representation of the entire *SeSAm* simulation model, which can be used independently, too. Simulation aspects are discussed in Chapter *11*.

The mapping of the *AAPL* model on the *SeSAm* simulation model is shown in Figure *12.11*.

epubli, ISBN 9783746752228 (2018)

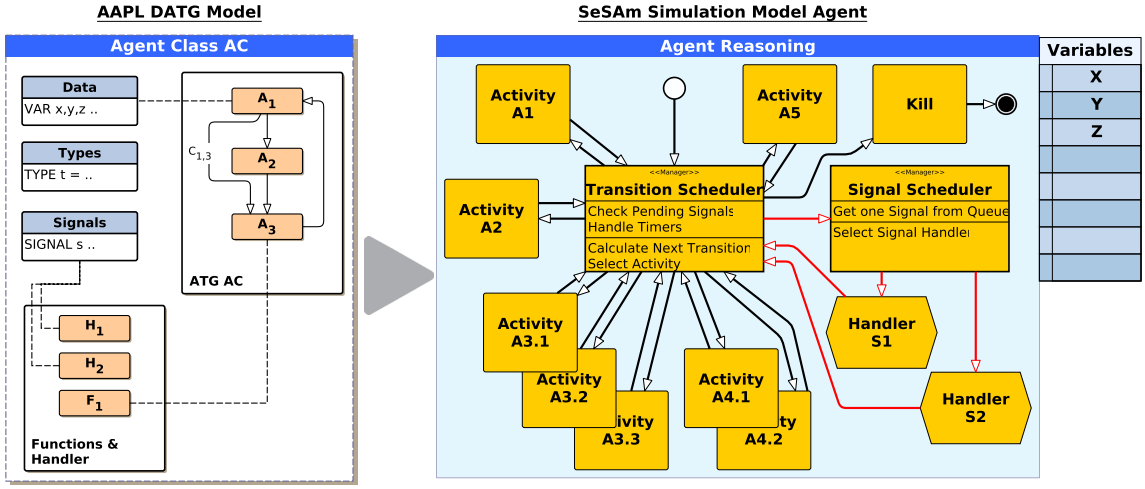

Fig. 12.11 *Simulation of AAPL based agents in SeSAm*

12.3.2 Programmable Agent Platform Synthesis

The programmable platform approach (the architecture was discussed in Chapter 7) is close to a traditional hardware-software co-design, though the agent processing platform (AVM) is either generic in terms of the agent behaviour classes, or optimized for a set of agent behaviour classes. In particular, the main synthesis part is the generation of efficient Agent Forth machine code form the *AAPL* specification. A detailed synthesis flow diagram is shown in Figure 12.12. The Agent Forth Programming Language (*AFL*) is an intermediate representation of the agent behaviour derived from *AAPL*, used finally to compile the machine instruction subset *AML*, which can be directly executed by the AVM.

The *AAPL* source code, which is commonly partitioned in different agent classes and files, is parsed by the parser generating an AST in the compiler database. Each AST is analysed and checked for model specification and operational semantics compliance.

Additionally, symbol tables are created and updated and references in the AST are linked to symbols. The entire MAS is analysed to derive design parameters, e.g., agent classes and subclasses analysis, computation of the data complexity, data types and sizes, tuple-space configuration, agent interaction, and estimated computational complexity, required for further machine code and platform synthesis.

The software compiler transforms the *AAPL* programming model first in an *AFL* program, which is eventually compiled to the machine instruction subset *AML*.

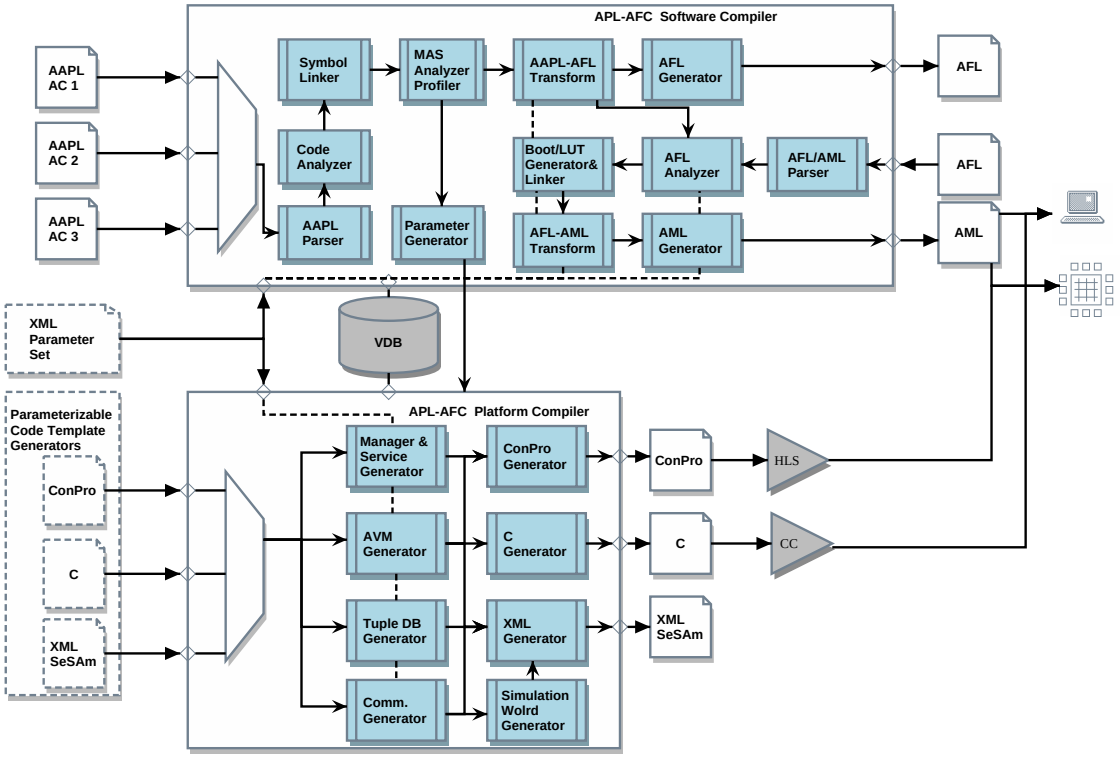

Fig. 12.12 *Software and Hardware Synthesis flow for the programmable Agent Forth processing platform.*

The *AFL* program code can be used as a secondary low-level MAS design entry, too. The AFL compiler part must generate a boot and LUT section for the program code frame. The AML program is a linear list of machine instructions without sub-structures and control environments like loop blocks. Most high-level *AFL* instruction are transformed to sequences of *AML* instructions by using transformation rule patterns, discussed below.

The platform compiler must generate the virtual machines capable to process the agent programs, the pipelined queue interconnection network, the local and global data memories, and the agent and communication managers. The communication protocol implementation is rather simple in this platform approach. It requires only the encapsulation of program code frames in messages send to a neighbour node. No routers are required.

AFL-AML Synthesis Rules

A composition with only a small set of special *AFL/AML* instructions provides agent creation, forking, migration, and modification based on code morphing, which is directly supported by the AVM.

The mapping of the operational semantics of the ATG agent behaviour and *AAPL* programming model on AFL programs and finally the transformation to AML machine code is primarily performed by the *AFC* compiler with the following synthesis rules.

Common Rules

The machine language AML represents only a sub-set of the Agent Forth programming language AFL. Common mapping rules are used to implement high-level functions with AML, shown in Table *12.1*. Some instructions are expensive to implement with AML, for example, `2swap`.

AFL	*AML*
`abs`	`DUP VAL(0) LT BRANCHZ(2) NEGATE`
`min`	`OVER OVER GT BRANCHZ(2) SWAP DROP`
`max`	`OVER OVER LT BRANCHZ(2) SWAP DROP`
`<>`	`EQ NOT`
`0=`	`VAL(0) EQ`
`0<>`	`VAL(0) EQ NOT`
`nip`	`SWAP DROP`
`tuck`	`SWAP OVER`
`-rot`	`ROT ROT`
`2dup`	`OVER OVER`
`2drop`	`DROP DROP`
`2swap`	`TOR ROT ROT TOR ROT ROT`
`2over`	`PICK(4) PICK(4)`
`2>r`	`SWAP TOR TOR`
`2r@`	`FROMR FROMR OVER OVER TOR TOR SWAP`
`2r>`	`FROMR FROMR SWAP`
`i`	`ONE RPICK`

Tab. 12.1 *AFL-to-AML synthesis rules (VAL: value literal word)*

AFL	AML
j	VAL(2) RPICK
k	VAL(3) RPICK
kill	CLEAR EXIT
call(TRANS)	VAL(-1) VAL(bootsize-1) BRANCHL
out	VAL(0) OUT
mark	OUT
tryin	IN
in	VAL(0) IN QBLOCK
tryrd	RD
rd	VAL(0) RD QBLOCK
rm	VAL(-2) IN
t+	VAL(1) VAL(1) BLMOD
t-	VAL(0) VAL(1) BLMOD
t*	OVER 0 VAL(-1) VAL(0) VAL(1) BLMOD VAL(1) VAL(1) BLMOD
?exist	VAL(-2) RD
if *I1* then *I2*	BRANCHZ(*L*) *I1* *L*:*I2*
if *I1* else *I2* then *I3*	BRANCHZ(*L1*) *I1* BRANCH(*L2*) *L1*: *I2* *L2*: *I3*
do .. loop	TOR *L1*: FROMR OVER OVER GT BRANCHZ(*L2*) TOR .. FROMR ONE ADD TOR BRANCH(*L1*) *L2*: DROP DROP
begin .. until	*L1*: .. BRANCHZ(*L1*)
var <name> <datatype> [n]	VAR <#lut> <size> DATA
: <name> .. ;	DEF <#lut> <size> .. [EXIT]
:%trans .. ;	TRANS <#lut> <size> <boot:4> ..

Tab. 12.1 *AFL-to-AML synthesis rules (VAL: value literal word)*

Agent Creation using Code Morphing

New agent processes can be created by using code templates and the `create` statement, by forking the code and the state of a currently running process using the `fork` statement, or by composing a new agent class from the current process.

Creating new and forking child processes are implemented with the previously introduced `NEW`, `LOAD`; and `RUN` machine instruction sequences, defined in Equations *12.1* and *12.2*, respectively.

$$\left\lceil \frac{a_1\ a_2\ ..\ a_n\ n_{args}\ ac\ \text{create}}{\text{VAL}(0^{\text{noinit}})\ \text{NEW DUP TOR SWAP LOAD FROMR VAL}(0^{new})\ \text{RUN}} \right\rceil \tag{12.1}$$

$$\left\lceil \frac{a_1\ a_2\ ..\ a_n\ n_{args}\ \text{fork}}{\text{VAL}(0^{\text{noinit}})\ \text{NEW DUP TOR VAL}(-1^{\text{fork}})\ \text{LOAD FROMR VAL}(1^{\text{fork}})\ \text{RUN}} \right\rceil \tag{12.2}$$

Agent Migration using Code Morphing

Process migration requires the saving of the data and control state of the process in the frame and transition table boot sections. After the migration had happened, the code frame is fully re-initialized including the loading of the process parameters. This requires currently the storage of the process parameter values on the data stack. In the future the parameter loading should be wrapped with a dynamic block, which is disabled after first execution, but must be re-enabled before forking.

The migration is a two-stage process: the first stage is executed by the `MOVE` operation, the second by the `SUSPEND` operation, shown in Equation *12.3*.

$$\left\lceil \begin{array}{c} \dfrac{dx\ dy\ \text{move}}{\text{MOVE VAL}(-1^{\text{root}})\ \text{SETCF VAL}(1^{\text{codeoff}})\ \text{STOC TOR}} \\ \text{REF}(p_1)\ \text{FETCH}\ ..\ \text{REF}(p_n)\ \text{FETCH} \\ \text{VAL}(n)\ \text{FROMR VTOC VAL}(1^{\text{fullinit}})\ \text{SUSPEND} \end{array} \right\rceil \tag{12.3}$$

12.4 The Agent Simulation Compiler SEMC

The MAS simulation techniques introduced in Chapter *11* using the Agent-based *SeSAm* simulator are carried out with complex simulation models. The specification of complex simulation models require a textual programming language representation of the simulation model (i.e., agents, worlds, resources, ..). But the *SeSAm* simulator offers a GUI and graph based modelling approach only, unsuitable for large simulation models like them used in this work.

For this purpose the *SEM* programming language was invented, compiled by the *SEMC* compiler to an *XML* simulation model, which can be directly read by the simulator. The block diagram of the compiler and the synthesis flow in shown in Figure *12.13*.

Different *SEM* input files are processes producing one simulation model file in *XML* format that can be directly imported by the *SeSAm* simulator. It includes *SeSAm* agent, resource, and world models.

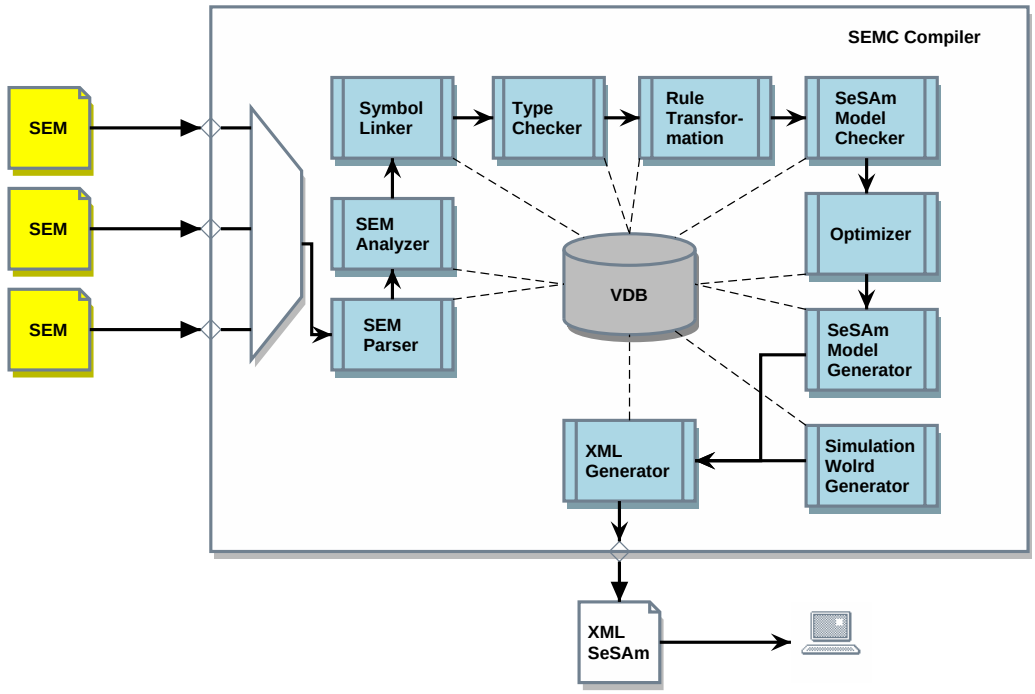

Fig. 12.13 *SEM Compiler synthesis flow and architecture*

12.5 ConPro SoC High-level Synthesis

Today there is an increasing requirement for the development of System-On-Chip designs (SoC) using Application-Specific Digital Circuits, with increasing complexity, too, serving low-power and miniaturization demands. The structural decomposition of such a SoC into independent sub-modules requires smart networks and communication (Network-on-Chip, NoC) serving chip area and power limitations. Traditionally, SoCs are composed of microprocessor cores, memory and peripheral components. But in general, massive parallel systems require modelling of concurrency both on control- and data-path level. Digital logic systems are preferred for exploration and implementation of concurrency. Traditionally, digital circuits are modelled on hardware behaviour or gate level, but usually the entry point for a reactive or functional system is the algorithmic level. The Register-Transfer-Logic (RTL) on architecture and hardware level must be derived from the algorithmic level, requiring a raise of abstraction of RTL [ZHU01].

With increasing complexity, higher abstraction levels are required, moving from hardware to algorithmic level. Naturally imperative programming languages are used to implement algorithms on program-controlled machines, which process a sequential stream of data- and control operations. Using this data-processing architecture, a higher-level imperative language can be simply mapped to a lower-level imperative machine language, which is a rule-based mapping, automatically performed by a software compiler. But in circuit design, there is neither an existing architecture nor an existing low level language that can be synthesized directly from a higher level one. An imperative programming approach provides both abstraction from hard- ware and direct implementation of algorithms, but usually reflects the memory-mapped von-Neumann computer architecture model. Another important requirement of a programming language in circuit design (in contrast to software design) is the ability to have fine-grained control over the synthesis process, usually transparent.

Using generic memory-mapped languages like *C* makes RTL hardware synthesis difficult due to the transparency of object references (using pointers) preventing RTL mapping. Additionally, concurrency models are missing in most software languages. There are many attempts to use *C*-like languages, but either with restrictions, prohibiting anonymous memory access with pointers, or using a program-controlled (multi-) processor architecture with classical hardware-software-co-design, actually dominant in SoC-Design. But SoC-designs using generic or application-specific processor architectures complicate low-power designs and concurrency is coarse grained.

One example is *PICO* [KAT02], addressing the complete hardware design ?ow targeting SoC and customizable or configurable processors, enhanced with custom-designed hardware blocks (accelerators). The RTL level is mod-

epubli, ISBN 9783746752228 (2018)

elled with *C*. The program-controlled approach with processor blocks enables soft- ware compilation and unrestricted C (functions, pointers) but lacks support of true bit-scaled data objects.

Another example is *SPARK* [GUP04], a *C*-to-*VHDL* high-level framework, currently with the restrictions of no pointers, no function recursion, and no irregular control-flow jumps. It is embedded in a traditional hardware-software-co- design flow. It is based on speculative code motions and loop transformations used for exploration of concurrency. SPARK generates pure RTL. Only a single-threaded control flow is provided.

Though *SystemC* provides many features suitable for higher-level synthesis, it is primarily used for simulation and verification, and only a subset can be synthesized to circuits. True bit-scaled data types are supported. Concurrency can be modelled using threaded processes, for example used in *Forthes* commercial synthesis tool *Cynthesizer* [COU08]. Inter-process communication is modelled on transaction level (TLM). *SystemC* provides a high- level-approach to model hardware behaviour and structure, rather than algorithms.

None of these approaches fully satisfy the requirements for pure RTL circuit design while using C-based languages, especially providing a consistent hardware, software, and concurrency model.

Efficient hardware design requires more knowledge about objects than classical languages like *C* can provide, for example, true bit-scaled registers, access, and implementation models on architecture level (for example single port versa dual port RAM blocks, static versa dynamic access synchronization). The generic software approach only covers the implementation of algorithms, but in hardware design the synthesized circuit must be connected to and react with the outside world (other circuits, communication links and many more), thus there must be a programming model to inter- face to hardware blocks, consistent with the imperative programming model. Furthermore, there must be a way to easily implement synchronization always required in presence of concurrency (at least on control path level). A multiprocess model, established in the software programmer community, provides a common approach for modelling parallelism, which is the preferred approach to implement and partition reactive systems on algorithmic level.

The *ConPro* programming language and synthesis [BOS10A][BOS11A] addresses the above described issues and introduces a design-methodology of SoCs using the concurrent multiprocess model and the advanced behavioural programming language discussed in detail in the Sections *5.4* and *5.8*.

The synthesis flow is defined by a set of rules χ shown in Equation (*12.4*). Each set consists of subsets, which can be selected by parameter settings (for example, scheduling like loop unrolling, or different allocation rules) on programming block level.

epubli, ISBN 9783746752228 (2018)

$$\text{Synthesis} : CP \xrightarrow{\chi^1} AST \xrightarrow{\chi^2} \left\{ \begin{array}{l} \mu CODE \xrightarrow{\chi^3} \left\{ \begin{array}{l} RTL \xrightarrow{\chi^4} VHDL \\ C \,/\, OCaML \end{array} \right. \\ C \,/\, OCaML \end{array} \right. \tag{12.4}$$

12.5.1 Synthesis Flow

The processing architecture of the *ConPro* compiler and the synthesis flow is shown in Figure 12.14. The synthesis process is a traditional software compiler flow with an intermediate representation (Synthesis layer 1, preserving parallel processing and CSP related features). There is a broad range of backends supporting software output, too (Synthesis layer 3).

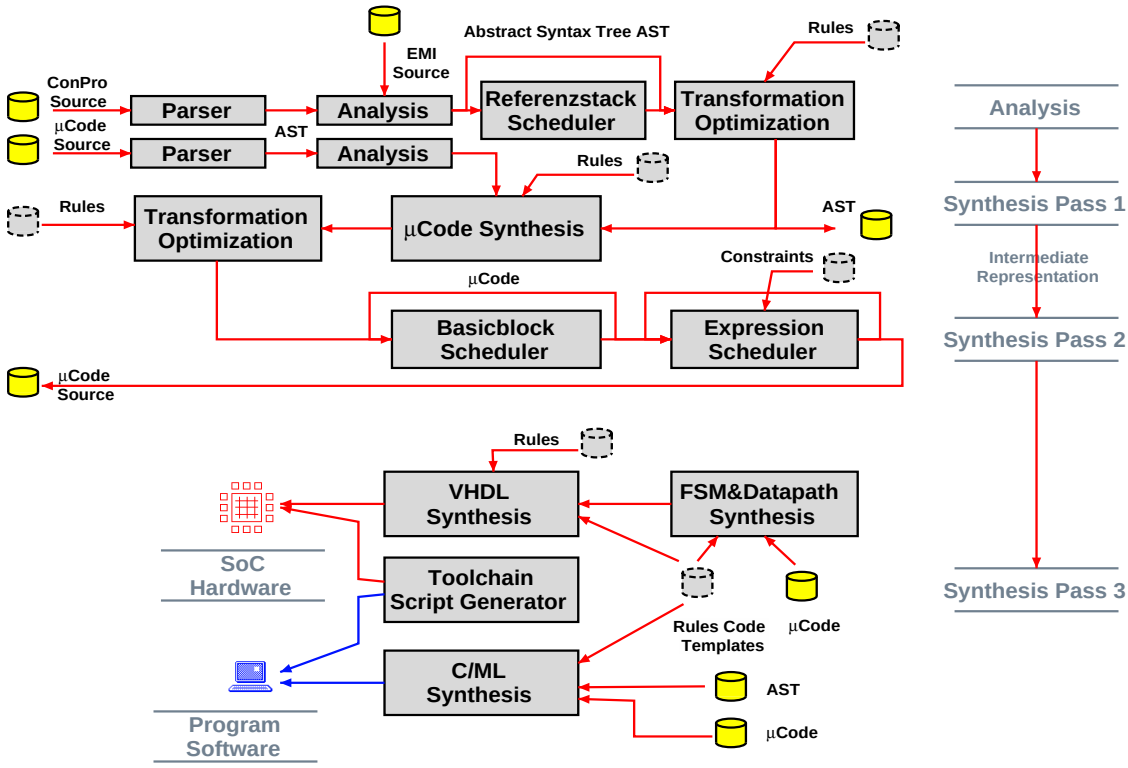

Fig. 12.14 *Design flow using the high-level synthesis framework ConPro that maps the parallel CCSP programming model to SoC-RTL hardware and alternative software targets.*

The synthesis of RTL digital logic circuits from high-level imperative *ConPro* program sources passes different phases:

1. First, the source code is parsed and analysed. For each process, an abstract syntax graph (AST) preserving complex statements is built. Global and local objects are stored in symbol tables (one globally for module level, and one for each process level).
   First optimizations are performed on the process instruction graph, for example, constant folding and dead object checking, and elimination of those objects and superfluous statements.
   Several program transformations (based on rules and pattern matching) are performed, for example inference of temporary expressions and registers.
   A symbolic source code analysis method, called reference stack scheduler [KU92], examines (local) data storage objects and their history in expressions. The reference stack scheduler analyses the evaluation of data storage expressions with an expression stack, one for each object.
   The reference stack transforms a sequence of storage assignments with expressions $\kappa=\{\Theta\leftarrow\varepsilon_1, \Theta\leftarrow\varepsilon_1, ..\}$ of a particular storage object Θ to a sequence of immutable and unique symbolic variables $\omega_i : \{\omega_1\leftarrow\varepsilon_1, \omega_2\leftarrow\varepsilon_1 , ..\}$. The aim is to reduce statements (using backward substitution and constant folding) and superfluous storage. The reference stack scheduler has an ALAP scheduling behaviour.
2. After the analysis and optimization on instruction graph level, the complex instructions (ranging from expressions to loops) of the AST are transformed into a linear list of μCode instructions. The μCode level is an intermediate representation of the program code, used in software compilers, too, though no architecture-specific assumption is made on this level, except constraints to the control flow. The μCode can be exported and imported, too. This feature enables a different en- try level for other programming language front-ends, for example, functional languages [SHA98].
   This intermediate representation allows more fine-grained optimization, allocation, and scheduling. The transformation from the syntax graph to μCode infers auxiliary instructions and register (suppose for-loops that require initialization, conditional branching, and loop-counter statements).
   Parallelism on data path level is provided by a bind instruction that binds multiple instructions to one execution step or one FSM state

(one time unit).
The transformation is based on a set of rules $\chi_{\kappa\to\mu}$, consisting of default rules and user-selectable rules (constrain rules). This is the first phase of architecture synthesis by replacing the paradigms of the source language with paradigms of the target machine, in this case a FSM with statements mapped on states and expressions mapped to the data path (RTL). Additionally, the first phase of allocation is performed here.
Data path concurrency is explored either by user-specified bounded blocks or by the basic block scheduler. This scheduler partitions the μCode instructions into basic blocks. These blocks have only one control path entry at the top and one exit at the tail. The instructions of one basic block (called major block) are further partitioned into minor blocks (containing at least one instruction or a bounded block). From these minor blocks data dependency graphs (DDG) are built. Finally, the scheduler selects data-independent instructions from these DDGs with ASAP behaviour.

3. After the first synthesis level, the intermediate *μCode* is mapped on an abstract state graph RTL using a set of rules $\chi_{\mu\to\Gamma\Delta}$, again consisting of default and user-selectable rules. A final conversion step emits *VHDL* code. This design choice provides the possibility to add other/new hardware languages, like *Verilog*, without changing the main synthesis path.
The rule set determines resource allocation of temporary registers and functional blocks providing different allocation strategies: shared versa non-shared objects and flat versa shared functional operators and inference of temporary registers. Shared registers and functional blocks introduce signal selectors inside the data path.
The RTL units are partitioned into a state machine FSM (two hardware blocks, one transitional implementing the state register and one combinational implementing the state switch network), providing the control path, and the data path (consisting of transitional and combinational hardware blocks, implementing functional operators, access of global resources and local registers).
4. Using the default set of rules, each μCode instruction (except those in bounded blocks) is mapped to one state of the FSM requiring one time unit (≥ 1 clock cycle execution time, depending on object guards and process blocking). Scheduling is mainly determined by the rule set $\chi_{\kappa\to\mu}$, rather than by $\chi_{\mu\to\Gamma\Delta}$.

Although no traditional iterative scheduling and allocation strategies are applied in this compiler flow, the non-iterative constraint-selective and rule-based synthesis approach provides inherent scheduling and allocation with strong impact from different optimizers. To summarize, there are different levels of scheduling and allocation:

Reference Stack Scheduler
Operates on syntax graph level and tries to reduce statements, functional operators, and storage, and has impact on scheduling and allocation.

Basic Block Scheduler
Operates on intermediate μCode level and tries to reduce operational time steps of statements and has only impact on scheduling.

Expression Scheduler
To satisfy timing and longest combinational path constraints, mainly clock-driven, complex, and nested flat expressions must be partitioned into sub-expressions using temporary registers and expanded scheduling. This scheduler has impact on both scheduling and allocation.

Optimizer
Classical constant folding, dead code and object elimination, and loop/branch-invariant code transformations further reduce time steps and resources (operators and storage).

Synthesis Rules
But finally the largest impact on scheduling and allocation comes from the set of synthesis rules $\chi = \chi_{\kappa\to\mu} \cup \chi_{\mu\to\Gamma\Delta}$

12.5.2 Microcode Intermediate Representation

ConPro source code is parsed and analysed in the first compilation stage into symbol lists and syntax graphs of complex instructions or environments for each process. In the secondary compilation stage, the process instruction graph structure is flattened in a linear list of microcode program instructions (μ*CODE*).

The main advantage of this approach is the simplified processing of a linear list of simple instructions from a small set rather than a complex graph structure with different complex programming environments and control statements, for example for-loops. Furthermore, this micro-code approach offers additionally the synthesis and implementation of microprocessor cores with a more common traditional hardware/software co-design, which is not considered here, but enables future extensions of the synthesis flow.

The set of μ*CODE* instructions are summarized in Table *12.2*.The first two instructions relate to the data path, the jump instructions relate to the control path. The function application instruction is required for the access of

Abstract Data Type Objects (ADTO) and their implementation. The ADTOs are commonly global resources that are shared by multiple processes. They require mutual exclusion synchronization and can provide higher level inter-process communication.

Instruction	*Description*	*Operand*	*Description*
`MOVE(dst,src)`	Data transfer dst ← src	`$immed.[i]`	Temporary symbolic variable used immediately in the next instruction(s)
`EXPR(dst, op1, operation, op2)`	Evaluate expression and assign result to `dst`.	`$temp.[i]`	Temporary storage register
`BIND(N)`	Bind the following *N* instruction to one bounded block.	`$alu.[i]`	Shared ALU
`JUMP(target)`	Branch to program position `target`	`DT{DT[N]}`	Data type
`FALSEJUMP( cond:target)`	Branch to program position `target` if `cond` is false	`ET{DT[N]}`	Expression data type
`FUN(name[,arg])`	Object operation or function application	`TC{DT[N]}`	Data type conversion
label:	Symbolic instruction (branch target) label		
`SPECIAL( instruction)`	Internal usage		
`NOP`	No operation place holder		

Tab. 12.2 *Language syntax of μCODE instructions: Left() instructions, (Right) operand formats*

The special and label instruction are auxiliary instructions. The bind instruction provides a way to bind data path instructions, which should be executed concurrently, either specified explicitly on programming level or implicitly derived by a scheduler and optimizer exploiting parallelism on data path level.

One obvious disadvantage of this approach is the lost of parallelism inside a process block, because each instruction or operation occupies at least one time step. To preserve parallelism on process instruction level, the bind instruction was introduced. This instruction binds the *N* following data path instructions into a group, and all instructions of the group are executed in one time step concurrently (perhaps higher number of clock cycles required due to guarding and blocking constraints).

Operands of μ*CODE* instructions can be registers, variables, signals, or temporary (symbolic) variables with additional information about data (DT) and expression types (ET) attached (sub-typing), required for type and data width conversion (TC) of operands to target object (LHS) types. More information are available about locality and LHS (write reference) or RHS (read reference) attributes, and if a shared ALU is used for an expression evaluation. The μ*CODE* instructions provides initial scheduling and allocation information for the following RTL synthesis including expression transformations.

12.5.3 Synthesis Rules

The simplest core set of synthesis rules applied to source code in the HLS is summarized below.

Rule P: Process

Each sequential *ConPro* process is mapped on one RTL block, with a FSM offering the control flow of the computation and a behaviour given by a state-transition-graph $\Gamma=(\Sigma,\psi)$, and a data path Δ. The set of control states is $\Sigma=\{\sigma_1, \sigma_2, ..\}$, and the set of control state transitions is *E*.

The entire RTL block is composed of combinational and transitional hardware blocks, $HW=(\Phi \cup \Pi)$.

Rule SI: Storage

There are only registers $\Re$ with independent data widths in the range $[1..W_{max}]$ bits.

```
reg name: DT[width];
```

Rule SII: Storage Resource Sharing and Allocation

There is no resource sharing: For each register $r \in \Re$ (which is used) a transitional hardware block τ is allocated.

$\texttt{FUN ALLOCATE}^{\texttt{STORAGE}} : \{r_1, r_2, .. \mid r_i \in \Re\} \rightarrow \{\tau_1, \tau_2, .. \mid \tau_i \in \Pi\}$

Rule SIII: Statement Scheduling

Each instruction *i* from the set of elementary statements κ is mapped on one state σ of the control flow $\Gamma=(\Sigma,\psi)$ of the FSM. The set of elementary statements consists of κ = {*Assignment*, *Expression Evaluation*, *Conditional Branch*, *Unconditional Branch*}.

FUN SCHEDULE : $\{i_1, i_2, .. \mid i_i \in \kappa\} \rightarrow \{\sigma_1, \sigma_2, .. \mid \sigma_i \in \Sigma\}$

Rule SIV: Data path

The data path Δ is decomposed to relation with the control state set ϕ.

Rule SV: Expression Resource Sharing and Allocation

There is no resource sharing. For each functional operator $op \in \aleph$ (which is used) a combinational hardware block ϕ is allocated. The set of operators consists of $\aleph$={`+,-,*,/,and, or, not, shiftl, shiftr`}.

FUN ALLOCATE^{EXPR} : $\{op_1, op_2, .. \mid op_i \in \aleph\} \rightarrow \{\phi_1, \phi_2, .. \mid \phi_i \in \Phi\}$

Rule SVI: Complex Statements

Complex statements $i \in \varepsilon$, for example, loops, are decomposed in a sequence of elementary statements using a synthesis decomposition rule χ: $\varepsilon_i \rightarrow \{i_1, i_2, .. \mid i \in \kappa\}$. The set of complex statements consists of ε={*If-Then-Else*, *Case-Select*, *For-Loop*, *While-Loop*, ..}.

The transformation rules for some complex statements are shown in Table *12.3* below.

Statement	*Transformation*
for cn_i = a to b do B_i	ALLOCATE: cn_i ➡ r_i ➡ τ_i TRANSFORM: $cn_i \leftarrow a$; $LOOP_i$: eval(cn_i < b)?, falsejump $EXIT_i$; B_i; $cn_i \leftarrow cn_i + 1$; jump $LOOP_i$; $EXIT_i$: ...
if e_i then $B_{i,1}$ else $B_{i,2}$	TRANSFORM: EVAL(e_i)?, falsejump $LB_{i,2}$; $B_{i,1}$; jump $EXIT_i$; $LB_{i,2}$: $B_{i,2}$; $EXIT_i$: ...
match e_i with when v_1 : $B_{i,1}$ when v_2 : $B_{i,2}$ ... when others : $B_{i,n}$	TRANSFORM: eval(e_i=v_1)?, falsejump $LB_{i,2}$; $B_{i,1}$; jump $EXIT_i$; $LB_{i,2}$: eval(e_i=v_2)?, falsejump $LB_{i,3}$; $B_{i,2}$; jump $EXIT_i$; ... $LB_{i,n}$: $B_{i,n}$; $EXIT_i$: ...

Tab. 12.3 *Synthesis transformation for complex ConPro statements. Notation is B: statement blocks, μCODE: x ← y; ➡ MOVE(y,x); eval(ε?), falsejump L; ➡ BIND(2); EXPR(t,ε); FALSEJUMP(t,L)*

Rule SVII: States and Transitions

Each state σ_i has always a successor state σ_j with $j \neq i$, except j=i iff the current statement execution is blocked by a guard of a global object (satisfaction of the guard condition). A state transition can be conditional, given by the state transition function COND(σ_i, σ_j , *cond*) satisfying a precondition *cond* or unconditional given by the function NEXT(σ_i, σ_j) (execution of an ordered statement sequence).

FUN TRANSITION: $(\sigma_i,\sigma_j) \rightarrow \{\rho \mid \rho \in \{\text{NEXT}(\sigma_i, \sigma_j), \text{COND}(\sigma_i, \sigma_j, \textit{cond})\}\}$

There is a start and an end state (σ_0, σ_∞) assigned to each sequential process. There is at least one state transitions to another state outgoing from each state, except for the end state σ_∞, which has only an outgoing self transition.

Ex. 12.4 *µCODE synthesis from ConPro program snippet (nop instructions are removed during the µCODE optimization and state compaction)*

CONPRO

```
s <- 0;
for i = 1 to 10 do
begin
  t <- 1;
  s <- s + i;
  if s = 0 then
    t <- 2;
  end;
end;
```

MICROCODE

```
  data:
        register LOOP_i_0: int[8]
  code:
        i1_assign:
          move (s,0)
        i1_assign_end:
          nop
        i2_for_loop:
          move (LOOP_i_0,1)
        i2_for_loop_cond:
          bind (2)
          expr ($immed.[1],10, > =,LOOP_i_0)
          falsejump ($immed.[1],%END)
        i3_assign:
          expr (s,s,+,LOOP_i_0:I8)
          nop
        i3_assign_end:
          nop
        i4_branch:
          bind (2)
          expr ($immed.[1],s,=,0)
          falsejump ($immed.[1],i2_for_loop_incr)
        i4_branch_end:
          nop
        i2_for_loop_incr:
          bind (3)
          expr (LOOP_i_0,LOOP_i_0,+,1)
          nop
          jump (i2_for_loop_cond)
        i2_for_loop_end:
```

Advanced synthesis rules include basic block scheduling (on AST and *µCODE* level), reference stack based optimization (in AST level), and resource sharing (especially concerning temporary storage), discussed in the next Sections.

Rule SOI: Extended set of storage objects

The set of storage objects is extended with variables stored in RAM blocks and selected by their address

Rule SOII: Resource sharing

Inside processes local storage like temporary registers can be reused for several computations. Furthermore, operations (arithmetic, relational, boolean) can be shared with an ALU approach, using one or multiple ALU blocks.

Rule SOIII: State compaction

The set of control states and the state-transition graph of each process derived from statement sequences and complex statement decomposition can be reduced by merging control and data statements. This is mainly performed on *μCODE* level.

Rule SOIV: Optimization

Several traditional optimization techniques like rescheduling (Reference Stack approach), constant folding, and dead object/code optimization can improve SoC resource requirements and execution times significantly, including basic block parallelization, discussed in the next sections.

12.5.4 Reference Stack (RS) Optimizer and Scheduler

The symbolic RS method offers storage and control flow optimization on AST level using an automatic scheduling approach, which is divided into two passes:

1. Merging of expressions with ALAP scheduling behaviour on abstract syntax tree level to resolve constant and register/variable folding beyond the initial instruction boundaries. The reference stack method merges expressions to meta expressions as large as possible, leading to optimised results in constant folding. All assignments of registers and variables are delayed on this level as late as possible. Real data transfer with register/variable assignments are scheduled only on global block, branch and loop boundaries.
2. Post scheduling of these (large) meta expressions with control step assignments depending on the chosen expression scheduling and allocation model on intermediate microcode level. In the case of a flat model, the meta expressions are scheduled in one time step. In the case of a two-ary or shared ALU model, the expression consisting of N

two-ary operations is scheduled in N time steps with temporary registers transfers.

Basically the RS algorithm (explained in Definition *12.4*) transforms a sequence of modifications of mutable storage objects into a sequence of immutable symbolic variables, shown in Example *12.5*. The flush of storage assignments is delayed as late as possible. Control statements can trigger a flush depending on the occurrence of storage objects in conditional and loop body blocks. The algorithm in Definition *12.4* is advanced version adapted from [KU92].

Ex. 12.5 *Possible outcomes (right) of a source code block (left) using the reference stack analysis and optimization (case 2: y is not referenced after this block and has no side effects). Bottom: Transformation of mutable storage to immutable symbolic variables*

```
x ← a + b + 1;
y ← x + 1 + c;        ➡   ❶ y ← a + b + c;
x ← y - 2;                   x ← y;
                          ❷ x ← a + b + c;

DEF x0 = a + b + 1
DEF y0 = x0 + 1 + c
DEF x1 = y0 - 2                T(x)=[x1,x0]
```

Def. 12.4 ***RS Algorithm***

1. Each data object *x* is traced with its own reference stack *T*:

   $T(x)=[re_{i-1}; re_{i-2}; \ldots ; re_0]$

   (Left element is the top of the stack, *re* is a reference expression of type *stackelement*)

   Data objects x

   ```
   register, variable
   ```

   Reference Expressions

   ```
   TYPE stack = stackelement list
   TYPE stackelement =
     | RS_self(obj)
     | RS_expr(expr)
     | RS_branch(stack list)
     | RS_loop(stack)
   ```

| `RS_ref(`*stackelement* `reference list)`

2. An assignment

   $x_i \leftarrow \varepsilon(x_1, x_2, ...)$

   is delayed as late/last as possible (ALAP).
3. A new reference stack for object *x* is created on the first occurrence of *x* on LHS in an assignment:

   $x_i \leftarrow \varepsilon_0(x_1, x_2, ...) \Rightarrow T(x_i)$=[`RS_expr(`$\varepsilon_0$)]

4. A new reference stack for object *x* is created on the first occurrence of x on the RHS in an assignment or an expression:

   $T(x_i)$=[`RS_self(`*x*)]

5. Each new assignment to a storage object *x* updates the reference stack by pushing the new expression ε (RHS of assignment) on the stack *T*(*x*).

   $x_i \leftarrow \varepsilon_n(x_1, x_2, ...) \Rightarrow T(x_i)$=[`RS_expr(`$\varepsilon_n$), ..]

6. Each occurrence of a storage object in an expression is marked (in the AST) with a reference to the current top element of the reference stack for this element (with top=n-1 and n stack elements): $\uparrow T(x_1)_{top}$

   $\varepsilon_n(x_1, x_2, ...) \Rightarrow \varepsilon_n(x_1{:}\uparrow T(x_1)_{top}, x_2 : \uparrow T(x_2)_{top}, ...)$

   Inside branches or loops the `RS_ref(`[*stack references*]) is placed on the reference stack *T*(*x*) for objects only referenced but not modified inside branches or loops, which is an important information for scheduling decisions (an RS flush, see next step).

7. Flush all delayed assignments at the end of the outer scheduling block and if there are no control statements like branches and loops before. After a flush has happened, the top of the reference stack is now `RS_self`. Resolve dependencies of objects and stack top expressions, sort assignments before flushing in correct dependency order.
   Relocate all references `RS_ref` pointing to other stack elements in expressions in the flushed assignments.

epubli, ISBN 9783746752228 (2018)

Example

```
x ← 12;        ✎ delay assignment, T(x)=[12]
y ← x + 1;     ✎ delay assignment, T(y)=[x0+1]
x ← v;         ✎ delay assignment, T(v)=[v], T(x)=[v0,12]
a ← y - 1;     ✎ delay assignment, T(a)=[y0-1]

➡ flush all delayed assignments with stack relocation and
   constant folding:

y ← y0=x0+1=12+1=13;
a ← a0=y0-1=x0+1-1=12;
x ← x1=v0=v;
```

8. *Branches*

i. Evaluate all conditional blocks $B_{cond,\,i}$ of the branch, with total b different blocks, each for one branch condition, j=0…b-1. Each conditional block is handled with its own sub-stack $T_{Bcond,i}(x)$ with n as the number of conditional blocks:

$T(x)$= [RS_branch([T_{n-1};T_{n-2},..,T_0],..]

All objects modified inside the conditional block $B_{cond,\,i}$ get a

RS_branch([$T_{Bcond,1}$= [], .., $T_{Bcond,\,i}$=[RS_expr(ε),..], ..])

stack element on the top of respective sub-stack. All further modifications are stored in this RS_branch stacks. Objects x appearing the first time in expressions get an RS_branch([.., $T_{Bcond,\,i}$=[↑$T(x)_{top}$], ..]) stack element. The initial top element T_0 of the branch is either a reference RS_ref to a previous expression or RS_self. The last element is required in loop environments due to side effects.

ii. Objects only referenced in each branch, that means they were not modified and $T_{Bcond,i}$=[RS_self | RS_ref] | ∉ RS_expr for all i=0…b-1, are transformed into references to previous stack elements:

$T(x)$=[RS_branch([[RS_self],[RS_self];...]),T_{n-2} ,..]
➡
$T(x)$=[RS_branch([[RS_ref(↑T_{n-2})],[RS_ref(↑T_{n-2})];..]),T_{n-2} ,..]

iii. After all conditional blocks were evaluated, objects with RS_branch([T_{n-1}; T_{n-2};...;T_0]) and T_i≠ [RS_self | RS_ref] | ∈ RS_expr (modified inside the block) must be pushed before the branch instruction (expectation: all possible branches modify x) if there is

an Expression RS_expr before the actual branch stack, and at the end of the conditional block they were modified.

iv. Modify the reference stacks:

a.) Storage object *x* was not modified inside the branch, but referenced:

$T(x)$=[RS_branch([[RS_self],..]),RS_expr(ε_i),..]

➡

$T(x)$=[RS_ref(RS_expr(ε_i)),
RS_branch([[RS_ref(RS_expr(ε_i))], ..]),RS_expr(ε_i), ..]

b.) *x* was modified inside the branch:

$T(x)$=[RS_branch([[RS_expr(ε_j),RS_self], ..]),RS_expr(ε_i), ..]

➡

$T(x)$=[RS_self,
RS_branch([[RS_expr(ε_j),RS_self)], ..]),RS_expr(ε_i), ..]

9. *Loops*

i. Evaluate all objects appearing in loop condition expressions before the loop body block is evaluated.

ii. Evaluate the body block *B* of the loop.
Objects appearing the first time in loop body expressions (and the RHS of assignments) get a RS_loop([T_0]) top stack element with T_0=RS_self. All objects modified the first time in the loop block get a RS_loop(T_{loop}) expression on the top of stack. All further modifications are stored in this RS_loop(T_{loop}) expression.

$T(x)$=[RS_loop(T_{loop}=[..]), ..]

iii. Objects only referenced or appearing on the RHS in the loop body, that means T_{loop}=[RS_self] | $\notin$ RS_expr, are transformed in references to previous stack elements, and this reference is pushed to the stack, too:

$T(X)$=[RS_loop([RS_self]),T_{n-2}, ..]

➡

$T(X)$=[RS_ref($\uparrow T_{n-2}$), RS_loop([RS_ref($\uparrow T_{n-2}$)]),T_{n-2}, ..]

iv. After the loop block was evaluated, objects with RS_loop(T) and $T\neq$ [RS_self] (modified inside the block) must be pushed before the

loop instruction if there is an expression RS_expr before the actual loop, and inside at the end of the loop block, too.

v. Modify the reference stacks:

a.) Storage object *x* was not modified inside the loop body, but referenced:

T(*x*)=[RS_loop([RS_self]),RS_expr(ε_i), ..]

➡

T(*x*)=[RS_ref(RS_expr(ε_i)),
RS_loop([RS_ref(RS_expr(ε_i))]),RS_expr(ε_i), ..]

b.) *x* was modified inside the loop body:

T(*x*)=[RS_loop([RS_expr(ε_j),RS_self]),RS_expr(ε_i), ..]

➡

T(*x*)=[RS_self,
RS_loop([RS_expr(ε_j),RS_self)]),RS_expr(ε_i), ..]

12.5.5 Basic Block Scheduling and Data Path Parallelization

The basic block scheduler explores the control flow and groups data path statements (assignments) in the largest possible statement blocks, the basic blocks. A basic block has only one control flow entry (the first statement), and only one exit (after the last statement). Basic blocks can be analysed regarding the exploitation of parallel execution of the statements of a basic block by using data dependency graphs (DDG). The DDG of a basic block is an acyclic directed graph with nodes representing the operations (assignments) and edges representing the data dependency of operations, determining an execution order.

A DDG of a basic block can indeed consists of a forest of independent graphs, and graphs with multiple root elements, shown in Figure *12.15*.

Independent operations can be scheduled and executed in one cycle with a bounded block. The goal is to schedule *k* operations in a bounded block with *k*={1,..,*c*} and *c* is either unlimited (maximal) or a limit of the maximal number of parallel operations. The DDG is partitioned in levels used finally for the scheduling. All operations belonging to one level are independent and can be scheduled in one bounded block, shown in Definition *12.5*. The level partition is performed by computing the longest path from a root node to each deeper node. Each node gets a marking of the longest path from a root node reaching this node, which is the equal to the level.

It is assumed that the parallel execution of a set of instruction contained in a bounded block is atomic and each data transfer is only executed one time.

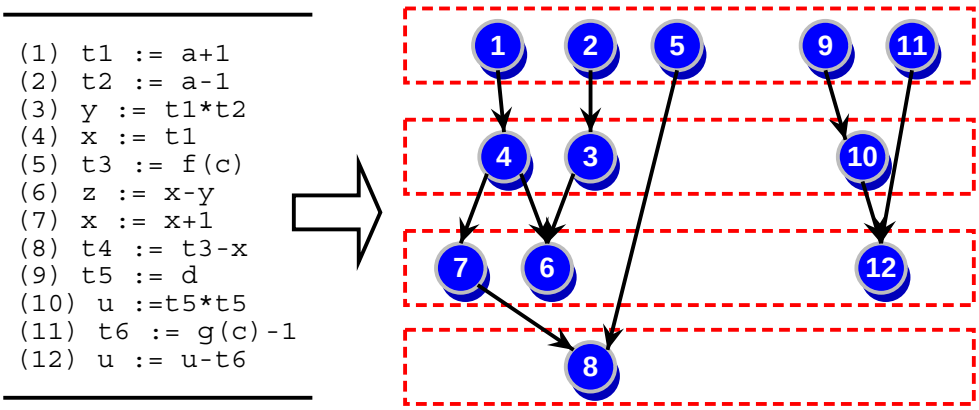

Fig. 12.15 *The derivation of a DDG forest from a sequence of assignments and a possible parallel scheduling requiring only four bounded block statements (and cycles).*

That means that expressions on the RHS of all assignments in a bounded block are first evaluated (with the old values of all storage objects referenced in the expressions).

Def. 12.5 *A.) Partition of an ordered list of instructions K={i_1; ..}, with i from the set of statements κ, in a set of basic blocks B=[b_1,b_2,..] consisting each of an ordered list of instructions (i;..). B.) The scheduling algorithm used to parallelize data flow independent sub-lists of instructions (i,..) mapped again to a flattened instruction list.*

```
FUN PARTITION: K=[i1;i2;..] → B=[b1;b2;..] IS
  b=[]; B=[];
  WHILE K ≠ [] DO
     k  ← HEAD(K), K ← TAIL(K);
    IF KIND(k) = DATA THEN b ← b ⊕ [k]
     ELSE B ← B ⊕ b, b ← [];


FUN SCHEDULE: B=[b1;b2;..] → K'=[(i1,i2,..);(..);..] IS
  K=[];
  ∀ b ∈ B DO
    DDG ← Build DDG forest from b;
    roots ← List of root nodes of DDG (nodes w/o predecessor);
    ∀ {op | op ∈ roots} DO
       Mark(op,1);
    maxLP ← 0;
    ∀ {op | op ∈ DDG\roots} DO
```

```
    LP ← 0;
    ∀ {op' | op' ∈ roots} DO
       LP ← Max(LP,|Path(op,op')|);
    Mark(op,LP);
    maxLP ← Max(maxLP,LP);
∀ {level | level ∈ {0..maxLP}} DO
    K ← K ⊕ [Bind({op ∈ DDG | Level(op) = level})];
```

12.5.6 RTL Synthesis and VHDL Model

Synthesis of Register-Transfer-Level design is enabled by a transformation of the intermediate microcode instruction list representation to FSM and RT Data path architecture blocks. The control and data path of the microcode instruction list is represented using a (linear) state list. A state list consists of concatenated state expressions, which can be considered as control flow statements containing data flow expressions for the particular state:

A state expression of a state transition list consists of:

state_name

State name string in the format: `S_i<instrid>_<instrname>_<subop>`

state_next

Different control statements (Control path of the RTL):

Next(label), Next_instr

Control flow is directed to labelled state <*label*>. The next instruction kind is an unresolved pointer to the next following state expression in the list.

Branch (data list, next$_0$, next$_1$)

Conditional branch. Data contains the boolean expression required for evaluating the conditional branch. Up to two possible state transitions are possible.

Select (data list, case_state list)

Multi-case non-hierarchical branching. Data contains the expression(s) to which each case value(s) is (are) compared. [*case_state=(data list, next_state)*]

state_data

Specification of the data path of the RTL block, e.g., different data transfer statements, mainly handling *VHDL* signal assignments:

Data_in(vhdl_expr)

Input arguments for an expression, e.g., the RHS of a data transfer statement, for example operands for the ALU.

Data_out (vhdl_expr)

Output from an expression or operational unit is directed to the LHS of a data transfer statement, for example the ALU output result or the output from a global object, acting only as a resource multiplexer (part of the combinatorial data path).

Data_trans(vhdl_expr)

LHS of an expression, transitional local register data transfer (part of the transitional data path).

Data_signal(vhdl_expr)

Signal assignments (state dependent), static driven.

Data_cond(vhdl_expr)

Conditional expression (expression transformed *VHDL* If/Case).

Data_top(vhdl_expr), Data_top_def (vhdl_expr)

Entity top-level VHDL expressions and architecture definitions.

Data_def(vhdl_expr,vhdl_expr), Data_def_trans (vhdl_expr,vhdl_expr)

Default signal assignments in combinatorial and transitional data path.

VHDL Output

Finally, *VHDL* entities are created from the state transition list (there is one list for each sequential process). Additional *VHDL* entities are created for modules. Global objects are implemented in the module top-level entity, shown in Figure *12.16*.

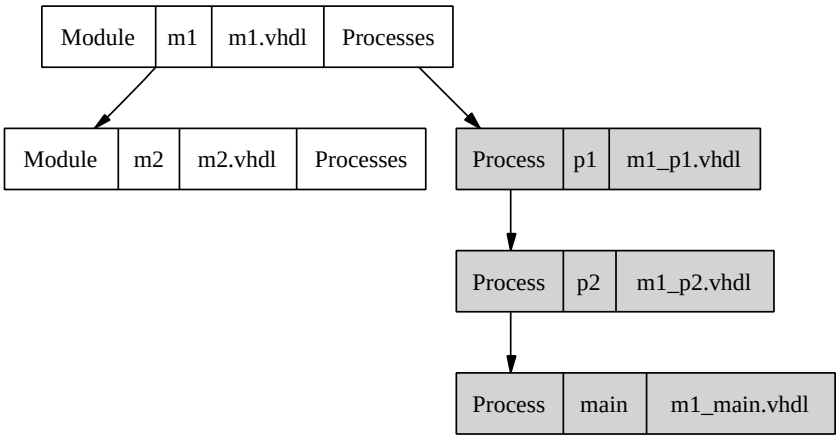

Fig. 12.16 *ConPro module hierarchy and created VHDL components.*

The following example poses the outline of the synthesized *VHDL* from a small *ConPro* design consisting of three processes and two shared objects, a register array and a semaphore used to synchronize the three processes.

Ex. 12.6 *A small ConPro design and excerpt of the synthesized VHDL*

ex.cp

```
open Core; open System; open Process; open Semaphore;

array data_in,data_out: reg[100] of int[16];
reg mon:int[16];          export mon;

object sem: semaphore with init=0;

function f(x:int[16]) return(y:int[16]): begin
  y <- x*x;
end;

process consumer1: begin
  sem.down();
  for i = 0 to 50 do begin data_out.[i] <- f(data_in.[i]); end;
  sem.up();
end;

process consumer2: begin
  sem.down();
  for i = 51 to 99 do begin data_out.[i] <- f(data_in.[i]); end;
  sem.up();
end;

process main: begin
  sem.init(0);
  for i = 0 to 99 do begin data_in.[i] <- i; end;
  consumer1.start(); consumer2.start(); sem.up();
  for i = 1 to 2 do begin sem.down(); end;
  for i = 0 to 99 do begin mon <- data_out.[i]; end;
end;
```

```
**********************************************************************
conpro ex.cp ➠
**********************************************************************
```

ex.vhdl

```
entity MOD_ex is
port(-- Connections to the outside world
  signal mon_RD: out std_logic_vector(15 downto 0);
  signal CLK: in std_logic;
  signal RESET: in std_logic
);
architecture main of MOD_ex is
```

```
  Process instances
  component ex_FUN_f
  port(..)
  component ex_consumer1
  port(..)
  component ex_consumer2
  port(..)
  component ex_main
  port(..)
  type ARRAY_data_out_TYPE is array(0 to 99) of
       signed(15 downto 0);
  signal ARRAY_data_out: ARRAY_data_out_TYPE;
  ..
  signal PRO_consumer1_ENABLE: std_logic;
  signal PRO_consumer1_END: std_logic;
  signal PRO_consumer1_main_START: std_logic;
  signal PRO_consumer1_main_GD: std_logic;
  ..
begin
```

ex_consumer1.vhdl

```
entity ex_consumer1 is
port(
  Connections to external objects, components and the outside
  world
  signal ARRAY_data_out_WR: out signed(15 downto 0);
  signal ARRAY_data_out_WE: out std_logic;
  signal ARRAY_data_out_GD: in std_logic;
  signal ARRAY_data_out_SEL: out std_logic_vector(7 downto 0);
  signal MUTEX_LOCK_FUN_f_LOCK: out std_logic;
  signal MUTEX_LOCK_FUN_f_UNLOCK: out std_logic;
  signal MUTEX_LOCK_FUN_f_GD: in std_logic;
  signal SEMA_sem_DOWN: out std_logic;
  signal SEMA_sem_UP: out std_logic;
  signal SEMA_sem_GD: in std_logic;
  signal REG_RET_FUN_f_y_RD: in signed(15 downto 0);
  signal PRO_FUN_f_CALL: out std_logic;
  signal PRO_FUN_f_GD: in std_logic;
  signal ARRAY_data_in_RD: in signed(15 downto 0);
  signal ARRAY_data_in_SEL: out std_logic_vector(7 downto 0);
  signal REG_ARG_FUN_f_x_WR: out signed(15 downto 0);
  signal REG_ARG_FUN_f_x_WE: out std_logic;
  signal PRO_consumer1_ENABLE: in std_logic;
  signal PRO_consumer1_END: out std_logic;
  signal conpro_system_clk: in std_logic;
  signal conpro_system_reset: in std_logic
);
end ex_consumer1;
architecture main of ex_consumer1 is
  Local and temporary data objects
```

```
  signal LOOP_i_0: signed(7 downto 0);
  State Processing
  type pro_states is (
    S_consumer1_start, -- PROCESS0[:0]
    S_i1_fun, -- FUN22215[ex.cp:19]
    S_i2_for_loop, -- COUNT_LOOP2877[ex.cp:20]
    S_i2_for_loop_cond, -- COUNT_LOOP2877[ex.cp:20]
    S_i3_fun, -- FUN78627[ex.cp:22]
    S_i4_assign, -- ASSIGN23890[ex.cp:22]
    S_i5_fun, -- FUN19442[ex.cp:22]
    S_i6_assign, -- ASSIGN58851[ex.cp:22]
    S_i7_fun, -- FUN83891[ex.cp:22]
    S_i2_for_loop_incr, -- COUNT_LOOP2877[ex.cp:20]
    S_i8_fun, -- FUN63052[ex.cp:24]
    S_consumer1_end -- PROCESS0[:0]
    );
  signal pro_state: pro_states := S_consumer1_start;
  signal pro_state_next: pro_states := S_consumer1_start;
..
  state_transition: process(..) begin
    if conpro_system_clk'event and conpro_system_clk='1' then
      if conpro_system_reset='1' or PRO_consumer1_ENABLE='0'
      then
        pro_state <= S_consumer1_start;
      else
        pro_state <= pro_state_next;
      end if;
    end if;
  end process state_transition;

  Process implementation
  Instruction Controlpath Block - The Leitwerk
  control_path: process(..) begin ..
    case pro_state is
      when S_consumer1_start => -- PROCESS0[:0]
        pro_state_next <= S_i1_fun;
      when S_i1_fun => -- FUN22215[ex.cp:19]
        if not((SEMA_sem_GD) = ('0')) then
          pro_state_next <= S_i1_fun;
        else
          pro_state_next <= S_i2_for_loop;
        end if;
      when S_i2_for_loop => -- COUNT_LOOP2877[ex.cp:20]
        pro_state_next <= S_i2_for_loop_cond;
      when S_i2_for_loop_cond => -- COUNT_LOOP2877[ex.cp:20]
        if CONST_I8_50 >= LOOP_i_0 then
          pro_state_next <= S_i3_fun;
        else
          pro_state_next <= S_i8_fun;
        end if;
      when S_i3_fun => -- FUN78627[ex.cp:22]
```

```
        if not((MUTEX_LOCK_FUN_f_GD) = ('0')) then
          pro_state_next <= S_i3_fun;
        else
          pro_state_next <= S_i4_assign;
        end if;
      when S_i4_assign => -- ASSIGN23890[ex.cp:22]
        pro_state_next <= S_i5_fun;
      when S_i5_fun => -- FUN19442[ex.cp:22]
        if PRO_FUN_f_GD = '1' then
          pro_state_next <= S_i5_fun;
        else
          pro_state_next <= S_i6_assign;
        end if;
      when S_i6_assign => -- ASSIGN58851[ex.cp:22]
        if ARRAY_data_out_GD = '1' then
          pro_state_next <= S_i6_assign;
        else
          pro_state_next <= S_i7_fun;
        end if;
      ..
      when S_consumer1_end => -- PROCESS0[:0]
        pro_state_next <= S_consumer1_end;
        PRO_consumer1_END <= '1';
  end process control_path;

  Instruction Datapath Combinational Block
  data_path: process(..) begin
    -- Default values
    SEMA_sem_DOWN <= '0';
    MUTEX_LOCK_FUN_f_LOCK <= '0';
    REG_ARG_FUN_f_x_WR <= to_signed(0,16);
    REG_ARG_FUN_f_x_WE <= '0';
    ARRAY_data_in_SEL <= "00000000";
    PRO_FUN_f_CALL <= '0';
    ARRAY_data_out_WE <= '0';
    ARRAY_data_out_SEL <= "00000000";
    ARRAY_data_out_WR <= to_signed(0,16);
    MUTEX_LOCK_FUN_f_UNLOCK <= '0';
    SEMA_sem_UP <= '0';
    case pro_state is
      when S_consumer1_start => -- PROCESS0[:0]
        null;
      when S_i1_fun => -- FUN22215[ex.cp:19]
        SEMA_sem_DOWN <= SEMA_sem_GD;
      when S_i2_for_loop => -- COUNT_LOOP2877[ex.cp:20]
        null;
      when S_i2_for_loop_cond => -- COUNT_LOOP2877[ex.cp:20]
        null;
      when S_i3_fun => -- FUN78627[ex.cp:22]
        MUTEX_LOCK_FUN_f_LOCK <= MUTEX_LOCK_FUN_f_GD;
      when S_i4_assign => -- ASSIGN23890[ex.cp:22]
```

```
                REG_ARG_FUN_f_x_WR <= ARRAY_data_in_RD;
                REG_ARG_FUN_f_x_WE <= '1';
                ARRAY_data_in_SEL <= (I_to_L(LOOP_i_0));
              when S_i5_fun => -- FUN19442[ex.cp:22]
                PRO_FUN_f_CALL <= '1';
              when S_i6_assign => -- ASSIGN58851[ex.cp:22]
                ARRAY_data_out_WR <= REG_RET_FUN_f_y_RD;
                ARRAY_data_out_WE <= '1';
                ARRAY_data_out_SEL <= (I_to_L(LOOP_i_0));
              ..
          end process data_path;

          Instruction Datapath Transitional Block
          data_trans: process(..) begin
            if conpro_system_clk'event and conpro_system_clk='1' then
              if conpro_system_reset = '1' then
                LOOP_i_0 <= to_signed(0,8);
              else
                case pro_state is
                  when S_consumer1_start => -- PROCESS0[:0]
                    null;
                  when S_i1_fun => -- FUN22215[ex.cp:19]
                    null;
                  when S_i2_for_loop => -- COUNT_LOOP2877[ex.cp:20]
                    LOOP_i_0 <= CONST_I8_0;
                  ..
                  when S_i2_for_loop_incr =>
                    -- COUNT_LOOP2877[ex.cp:20]
                    LOOP_i_0 <= LOOP_i_0 + CONST_I8_1;

ex_consumer2.vhdl
        entity ex_consumer2 is ..

ex_FUN_f.vhdl
        entity ex_FUN_f is
        port(
          -- Connections to external objects, components and the outside
world
          signal REG_RET_FUN_f_y_WR: out signed(15 downto 0);
          signal REG_RET_FUN_f_y_WE: out std_logic;
          signal REG_ARG_FUN_f_x_RD: in signed(15 downto 0);
          signal PRO_FUN_f_ENABLE: in std_logic;
          signal PRO_FUN_f_END: out std_logic;
          signal conpro_system_clk: in std_logic;
          signal conpro_system_reset: in std_logic
        );
        ..

ex_main.vhdl
        entity ex_main is
        ..
```

12.5.7 Dining Philosopher Example

Example *12.7* shows a part of the program code for the implementation of the dining philosopher problem. This system consists of five processes concurrently executing and implementing the action of the philosophers. They are defined using an array construct (line 13). The instructions of each process are processed sequentially, indicated by a semicolon after each instruction statement. The resource management of the forks is done with semaphores (abstract object type). Each philosopher process tries to allocate two forks, the left- and right-hand side forks, by calling the down method for each semaphore. If a philosopher process succeeds, it calls the (in-lined) function eat, and sets global registers (eating, thinking) simultaneously, indicated in the program code by using a colon instead of a semicolon (bounded instruction block). An event object *ev* is used to synchronize the start-up of the group. The philosopher processes waiting for the event by calling the await method (line 15). The event is woken up by the main process calling the `wakeup` method (line 41). All processes are started from the main process (line 40). Processes are treated as abstract objects, too, providing a set of methods controlling the process state.

Objects (like IPC) belong to a module, which have to be opened first (line 1). Each module is defined by a set of *EMI* implementation files providing all necessary information about objects of this module (like method declarations, object access, and implementation on hardware and software level).

Ex. 12.7 *Parts of a ConPro source code example: the dining philosopher problem implementation mapped on a multiple processes using semaphores for resource management.*

```
1   open Core; open Process; open Semaphore; open System; open Event;
2   object ev: event;
3   array eating, thinking: reg[5] of logic;
4   export eating,thinking;
5   array fork: object semaphore[5] with
6         Semaphore.depth=8 and Semaphore.scheduler="fifo";
7   function eat(n):
8   begin
9     eating.[n] <- 1,thinking.[n] <- 0;
10    wait for 5;
11    eating.[n] <- 0,thinking.[n] <- 1;
12  end with inline;
13  array philosopher: process[5] of
14  begin
15    ev.await ();   synchronize all processes
16    if # < 4 then  all processes with array index lower 4
17    begin
```

```
18     always do
19     begin
20       get left fork then right
21       fork.[#].down (); fork.[#+1].down ();
22       eat (#);
23       fork.[#].up (); fork.[#+1].up ();
24     end;
25    end
26    else
27    begin
28     always do
29     begin
30       -- get right fork then left
31       fork.[4].down (); fork.[0].down ();
32       eat (#);
33       fork.[4].up (); fork.[0].up ();
34     end;
35    end;
36  end;
37  process main:
38  begin
39    for i = 0 to 4 do
40      philosopher.[i].start ();
41    ev.wakeup ();   start the game ...
42  end with schedule="basicblock";
```

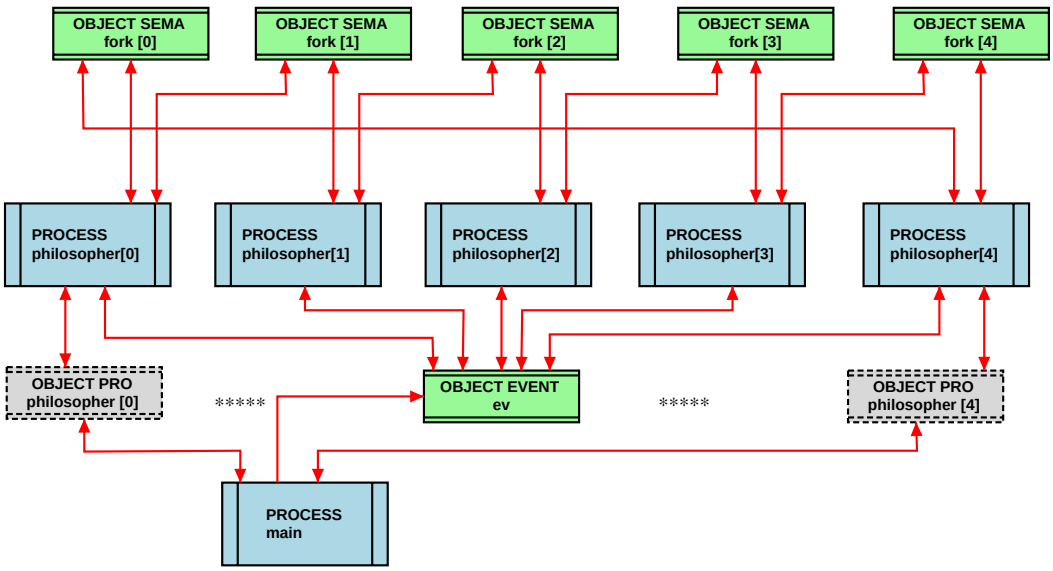

Fig. 12.17 *Process and inter-process-communication architecture of the dining philosopher problem implementation from Example12.7.*

The structural CCSP model is shown in Figure 12.17. Processes are objects and execution units, too. Each global shared object is guarded by a Mutex scheduler resolving concurrent access by serialization.

12.5.8 Complex ConPro SoC Design Case Study

Beside results of the agent platform synthesis presented in Chapter 6 a complete implementation of the adaptive routing *SLIP* protocol stack, introduced in Section 4.3.3, is shown here [BOS11A].

SLIP implements smart routing of messages with Δ-addressing of nodes arranged in an n-dimensional network space (line, mesh, cube). The network can be heterogeneous regarding node size, computation power, and memory. The communication protocol is scalable regarding network topology and size.

A node is a network service endpoint and a router, too, which must be implemented on each network node.

To summarize, the routing information is always kept in the packet, consisting of:

1. A header descriptor (*HDT*) specifying the address size class *ASC*, the address dimension class *ADC* (for example 2 is a two-dimensional mesh-grid);
2. A packet descriptor (*PDT*) with routing and path information, and finally the data part.

The *SLIP* protocol implementation must deal with the variable format of a message. *SLIP* was designed for low-resource System-On-Chip implementations using ASIC/FPGA target technologies, but a software version was required, too. To minimize the resource requirement, the hardware implementation will be usually limited in the dimensional and address space supporting only a subset of the message space, e.g. ***ADC***=2, ***ASC***=8 (that means δ=-128,..,127).

A node should handle several serial link connections and incoming packets concurrently, thus the protocol stack is a massive parallel system, and was implemented with the *ConPro* behavioural multiprocess model.

The programming model implementation with partitioning of the protocol stack in multiple processes executing concurrently and communicating using queues is shown in Figure 12.18.

Each link is serviced by two processes: a message decoder for incoming and an encoder for outgoing messages. A packet processor `pkt_process` applies a set of smart routing computation functions (`route_normal`, `route_opposite`, `route_backward`, applied in the given order until routing is possible), finding the best routing direction. Communication between processes is implemented with queues. There are three packet pools holding HDT, PDT and data

parts of packets. They are implemented with arrays. The packet processor can be replicated to speed up processing of packets.

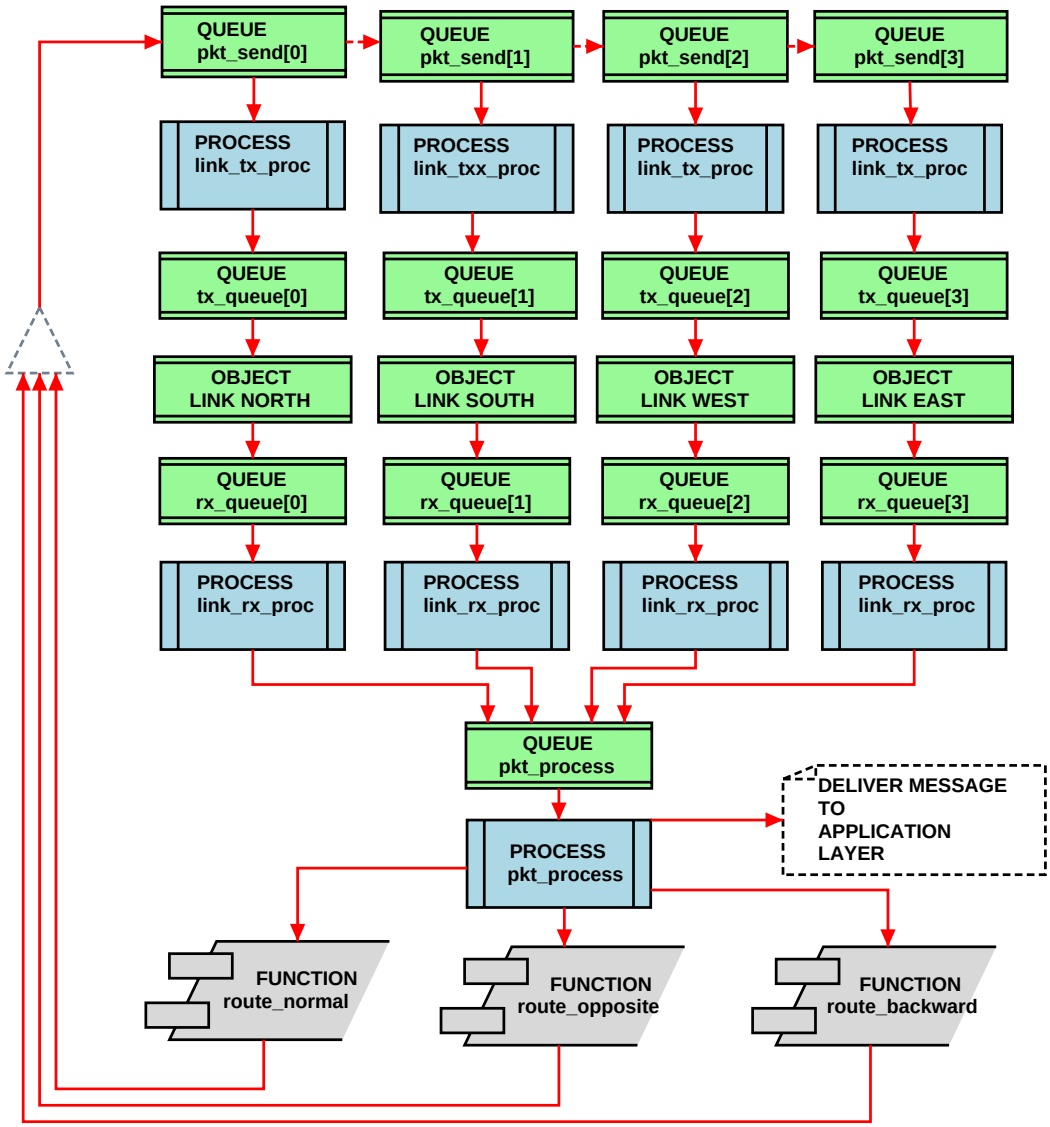

Fig. 12.18 *Process and inter-process-communication architecture of the SLIP protocol stack.*

A test setup consisting of the routing processing part of *SLIP* was implemented A. in hardware (RTL-SoC, gate-level synthesis with Mentor Graphics Leonardo Spectrum and SXLIB standard cell library), and B. in software (*SunOS*, *SunPro C* compiler). A packet with ADC=2, Δ=(2,3) and a link setup of the node L=(-y,-x) is received on the second link (-x) [L01] and is processed first by the `route_normal` rule (would require connected +x /+y links) [L03], and finally by the `route_opposite` rule [L04] forwarding the modified packet to the link_0 process [LA0].

Tables *12.4* and *12.6* show synthesis and simulation results for the hardware (HW) and the operational equivalent software (SW) implementation. They show low resource demands and latency. Different checkpoints Lxx indicate the progress of packet processing. Figures in brackets give the latency progress relative to the previous checkpoint.

Resource	*Variable[1]*	*Register[2]*
Registers [FF]	767	587
Area [gates]	12475	10758
Path delay [ns]	18	16
Synthesized Source CP → VHDL	1109 → 9200 lines	1109 → 7900 lines

Tab. 12.4 *Comparison of resources required for the HW implementation of the routing part of SLIP implemented with a packet pool: (1) variable array, (2) register array. ASIC synthesis was performed with Leonardo Spectrum software and SXLIB standard cell library.*

Two different storage resource models are compared: variable arrays with RAM blocks and register arrays. Surprisingly, the register storage model is more resource efficient than the RAM storage model. But this fact holds only for ASIC technologies, and not for FPGA technologies with predefined functional units and already contained on-chip block RAM resources. The register storage model leads to lower computational latency of the parallel packet processing due to the CREW access behaviour. The full design implementation with a *Xilinx* Spartan III (1000k eq. gates) FPGA and the ASIC synthesis is compared in Table *12.5*.

Resource	*FPGA*[1]	*ASIC*[2]
Registers [FF]	2925	15000
LUT [#]	11261/15360	-
Area [gates]	-	244 k gates ≈ 2.5mm^2 | 0.18µm
Conpro Source	4000 lines, 34 processes, 30 shared objects (16 queues, 2 timers)	
Synthesized VHDL	32000 lines	

Tab. 12.5 *Comparison of resources required for the full HW implementation of SLIP including simple application layer: (1) FPGA synthesis was performed with Xilinx ISE, (2): ASIC was performed with Leonardo Spectrum software and SXLIB standard cell library*

Checkpoint	*Clock Cycles*[1]	*Clock Cycles*[2]	*Machine Operations*
L01	104	102	60000
L03	113 (δ=9)	107 (δ=5)	60019 (δ=19)
L04	187 (δ=74)	148 (δ=41)	60796 (δ=777)
LA0	235 (δ=48)	184 (δ=36)	62305 (δ=1509)

Tab. 12.6 *Simulation results of the HW and SW implementation of the routing part of SLIP. HW: packet pool: (1) variable array, (2) register array, clock cycles. SW: SunPro CC, SunOS, USIII, CPU machine operations*

epubli, ISBN 9783746752228 (2018)

12.6 Further Reading

1. S. GUPTA, R. K. GUPTA, N. D. DUTT, and A. NICOLAU, SPARK: *A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Kluwer Academic Publishers, 2004, ISBN 1402078374
2. G. Ku, David C., DeMicheli, *High Level Synthesis of ASICs under Timing and Synchronization Constraints*. Kluwer Academic publishers, 1992, ISBN 9781475721171
3. R. Sharp, *Higher-Level Hardware Synthesis*, Springer Berlin, 2004, ISBN 3540213066