A Compositional Framework for Hardware/Software Co-Design *

A. Cau (cau@dmu.ac.uk), R. Hale (rhale@dmu.ac.uk), J. Dimitrov (jordan@dmu.ac.uk), H. Zedan (zedan@dmu.ac.uk) and B. Moszkowski (benm@dmu.ac.uk)
Software Technology Research Laboratory
SERCentre, De Montfort University, England

M. Manjunathaiah (manju@comlab.ox.ac.uk) and M. Spivey (m.spivey@comlab.ox.ac.uk)
Programming Research Group
Computing Laboratory
Oxford University, England

Abstract. We describe a compositional framework, together with its supporting toolset, for hardware/software co-design. Our framework is an integration of a formal approach within a traditional design flow. The formal approach is based on Interval Temporal Logic and its executable subset, Tempura. Refinement is the key element in our framework because it will derive from a single formal specification of the system the software and hardware parts of the implementation, while preserving all properties of the system specification. During refinement simulation is used to choose the appropriate refinement rules, which are applied automatically in the HOL system. The framework is illustrated with two case studies. The work presented is part of a UK collaborative research project between the Software Technology Research Laboratory at the De Montfort University and the Oxford University Computing Laboratory.

Keywords: HW/SW Co-Design, Refinement, Compositionality, Executable Specification, HDL

1. Introduction

Design and analysis of embedded, mixed hardware/software systems, such as PC cards and associated software drivers, is hard. A major reason for this is the ever increasing complexity of hardware and software systems coupled with the historical divide between hardware and soft-

* Funded by the U.K. Engineering and Physical Sciences Research Council (EPSRC) through Research Grants GR/M/32474 and GR/M/32481

ware design. Very often there are conflicting goals, and trade-offs have to be made to find the best compromise between them. Some typical aspects which have to be balanced are performance, cost, flexibility, distribution, power consumption, size and fault tolerance.

An important issue for correct co-design is the search for *a highly compositional lean formal approach* that crosses the hardware/software boundary and enables us to keep up with the fast growth in the complexity and variety of electronic devices and their associated software. By *compositional* approach we include any method by which the properties of a system can be inferred from properties of its components, without additional information about the internal structure of those components [13]. And by *lean* formal approach, we take the view that the method must be supported by automated tools that make the method more widely accessible to users. Such tools should support rapid prototyping, a compositional design process and formal verification.

In the hardware industry, *simulation* has often been considered synonymous with *verification*. The design process usually still consists of developing an implementation from an informal specification without the use of any formal design techniques. The hardware and software are then simulated for a number of inputs, an approach known as *co-simulation* [45, 1, 16]. Bugs discovered are removed and the simulation process is repeated over again.

However, formal verification cannot completely replace the existing simulation approach. This is because simulation provides more accessible tools for rapid prototyping and testing. What is needed is an approach where the design process is soundly based upon formal techniques, but includes integrated support for simulation. This combination would bring more reliability within an environment which is consistent with current practice. The provision of automated tools is absolutely essential for making a formal theory accessible to a wider audience. Such tools can assist even specialists who wish to develop and verify specifications or who are dealing with large systems with many details. Tools like Tempura [40, 24] and METATEM [3]) allow restricted but useful classes of logical formulas to be executed, which facilitates the debugging of specifications.

In this paper, we describe an integrated compositional framework, together with its supporting toolset, for hardware/software co-design. The co-design process is soundly based upon formal techniques, but also supports simulation. A unique characteristic of our framework is that it can validate and analyze system's behaviors within a *single* logical formalism, namely Interval Temporal Logic (ITL) and its executable subset, Tempura. An integrated suite of tools supporting our development strategy is provided: For simulation and analysis, AnaTempura is used while for formal verification and mechanical refinement we use an embedding of ITL in HOL [20]. Note that refinement is an interactive process and thus can't be fully automated.

1.1. Related Work

Our approach is inspired by existing co-design systems, such as SpecC [18], Polis [2] and LYCOS [36]. The traditional design flow is that a project starts with *Informal Specification*, also called requirement, which defines the behavior and the functionality of the product. Immediately after the specification, a designer should split the application into *hardware* and *software* [17]. Our work integrates formal methods into this design process. Our focus in this paper, however, is on refinement from a formal specification into a formal hardware part and a formal software part.

Many existing co-design systems, such as Polis [2], LYCOS [36], include some formal verification capability, which is most often achieved by use of an external tool, such as a model-checker. The model-checker can only be used when the design is already quite concrete and such an approach can not maintain the integrity of the whole design. In contrast, our approach enforces correctness of the design process by working entirely within a formal system.

There have been successful hardware/software verification efforts in academia and more recently in industry. The majority have used model-checking techniques [11, 34, 27], but also for example functional calculi [12, 30] and Abstract State Machines [5], and recently more powerful tools such as HOL [20] have been gaining ground.

As well known, one can use several abstraction levels when developing a system in Verilog HDL [31]. The Verilog Formal Equivalence project has been concentrating on different semantics for the different abstraction levels in Verilog [19]. Sagdeo and Thomas in [44, 47] give detailed design flows where the main stages are *Behavioral design* and *RTL design*. In [44, 47], a third abstraction level is included called *Gate Level Verilog* or *Structural design*. The reason we do not consider structural descriptions is that there are commercially available synthesis tools which transform RTL down to netlists and this step has already been automated, via the Synopsys synthesizer for example. The restrictions imposed on RTL specifications imply their synthesisability. However, behavioral descriptions, including event controls and high level language constructs, are generally not synthesisable.

According to the design flow given in [44], designers have to transform the *Behavioral* description into *RTL* using high level synthesis tools. At every step of the design process, simulations and tests are performed to check the correctness of the transformations with respect to the requirements. Although these tests can be automated to a considerable degree, there are many cases when testing only does not provide the necessary level of correctness. More often than not, crucial test cases are overlooked which, in the case of a critical system, may result in human lives and/or money being lost.

We have given formal semantics to Verilog in both *Denotational* (in the form of specification-oriented) and *Operational* terms [14, 15]. These two reflect the duality of the usage of specification languages, i.e., we need to both describe *properties* and *machines* which implement, or compute, these properties. We have used these semantics for Verilog to derive our refinement rules. We are building support for refinement by embedding it in the HOL [20] system [23]. Several other work on the semantics of Verilog exists namely at UNU/IIST [6, 33, 29, 28] and Cambridge [19].

There is increasing industrial interest in ITL, for example Verisity has adopted concepts from ITL in their *Temporal e* language [26]. IBM has introduced a temporal logic called *Sugar* [4] containing ITL-like operators which are targeted at making the logic more usable to design engineers.

1.2. Paper Organisation

In Section 2 we give an overview of our computational model which serves as an architecture for hardware/software co-design. In Section 3 we describe our specification language. The semantics for the HDL of choice Verilog is given in section 4. The refinement calculus will be given in Section 5 and two case studies are given in Section 6.

2. Framework for Hardware/Software Co-design

In this section we outline our framework for hardware/software co-design. Our framework is an integration of a formal approach with a traditional design flow. This section gives a brief discussion of the whole framework but the paper will give a detailed exposition of the key element of our approach: refinement.

The process of modeling a system, albeit sequential or concurrent, timed or untimed, needs a suitable computational model. We take the view that a computation defines mathematically an abstract architecture upon which applications will execute. A *system* is a collection of *agents* (which is our unit of computation), possibly executing concurrently and communicating (a)synchronously via communication links. Systems can themselves be viewed as single agents and composed into larger systems. Systems may have timing constraints imposed at three levels; system wide communication deadlines, agent deadlines and sub-computation deadlines (within the computation of an individual agent).

At any instant in time a system can be thought of as having a unique *state*. The system state is defined by the state variables of the system and, for concurrent system, by the values in the communication links. *Computation* is defined as any process that results in a change of system state. An agent is described by a computation which may transform a private data-space and may read and write to communication links during execution. The computation may have both minimum and maximum execution times imposed.

It is important to note that when we talk about *system* we do not make any distinction between software or hardware. We simply talk of a set of *agents* collaborating to achieve the desired behavior. Some of those agents may be realized (or implemented) in software and some in hardware.

Fundamental to our proposed investigation is that a synthesis and design methodology should start with a high-level abstract specification which describes the desired behavior(s) of the system under consideration. The target system is derived via design decisions made through *correctness preserving* refinement steps. Our proposed development strategy is depicted in Fig. 1 below.

The design process begins with a high-level abstract specification written in ITL. Properties of interest can be *compositionally* verified using the ITL's compositional proof rules in *assumption/commitment* style [41]. At this level we make no distinction between software or hardware. Using a sound refinement calculus, the ITL specification can then be refined into a set of Tempura modules (an executable subset of ITL) and simulated and analyzed (using AnaTempura, a part of the ITL Workbench). During this process, various design decisions are made, e.g. synchronous/asynchronous, sequential/parallel, etc.

This is followed by a 'module analysis' phase in which a set of quantitative and statistical data may be obtained (in [42] various techniques are given which can be utilized). Using existing work on partitioning [22] we split this Tempura set into two sets of modules, namely *Tempura-H* and *Tempura-S*. These are best realized in hardware and software implementation, respectively. The interface between these sets will depend on the target architecture and is formulated as, what we call an *interface theorem* which in turn can be verified compositionally using the ITL proof rules.

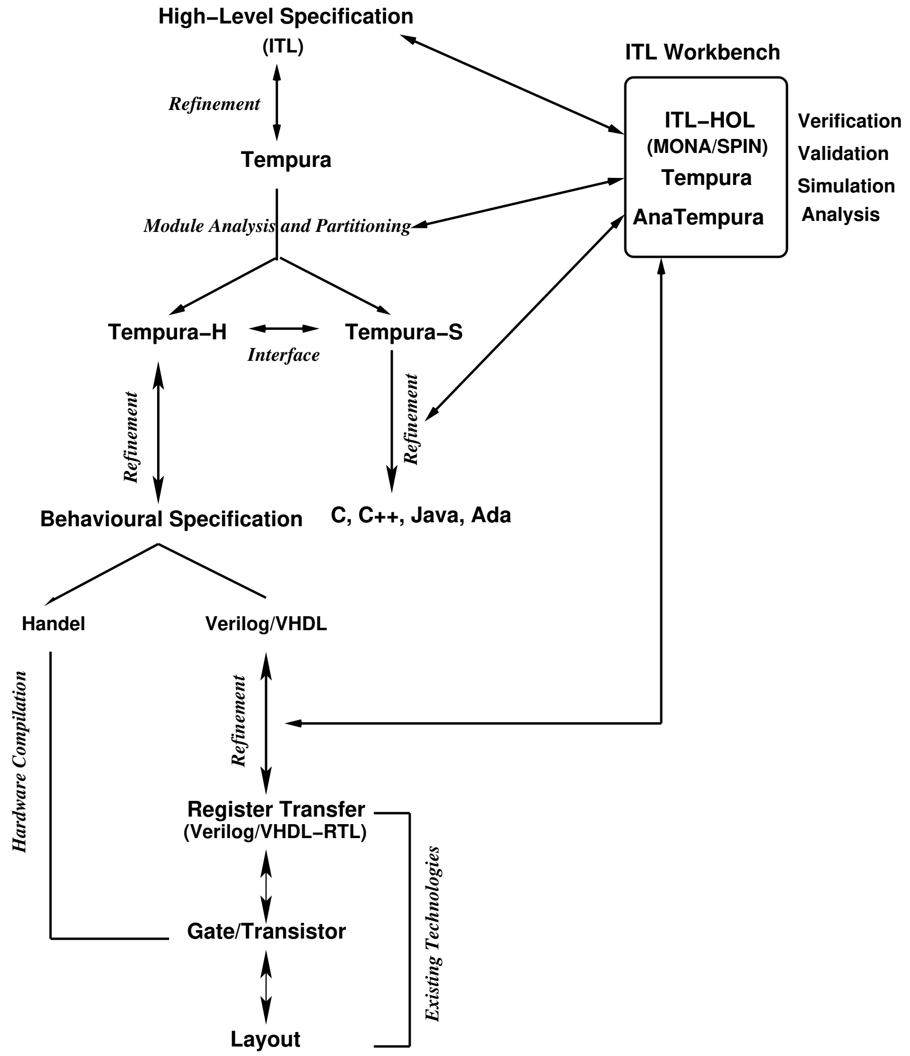

Figure 1. The development strategy

As depicted in Fig. 1, the abstraction gaps existing between *Behavioral*, *RTL* and *Gate levels* are bridged using sound refinement rules. For this to be realized, a unifying semantics for the various notations (used at each level) is needed. Such a unifying semantics is detailed in section 4 for our chosen HDL Verilog.

Using sound refinement/transformation rules, the modules in the *Tempura-S* set are transformed into software components written in popular languages, such as Java, C or C++ [8, 7]. Similarly, modules in the *Tempura-H* set are further refined into hardware description languages such as Verilog, VHDL or Handel [9]. As depicted in Fig. 1, a refinement calculus is also used to bridge the gap between the various abstraction levels in these technologies.

3. Specification Formalism

In this section we will introduce our specification formalism. As we mentioned earlier, our proposed approach is based on a single logical framework whose underlying logic is Interval Temporal Logic (ITL). In this section we provide a short description of the logic but a more detailed exposition may be found in [40].

ITL is a flexible notation for both propositional and first order reasoning about intervals (behaviors) found in descriptions of hardware and software systems. It can handle both sequential and parallel composition unlike most temporal logics. It offers powerful and extensible specification and proof techniques for reasoning about properties involving safety, liveness and timeliness.

ITL is a linear-time temporal logic with a discrete model of time. An interval σ in general has a length $|\sigma| \geq 0$ and a (in)finite, nonempty sequence of $|\sigma| + 1$ states $\sigma_0, \ldots, \sigma_{|\sigma|}$. Thus the smallest intervals have length 0 and one state. Each state σ_i for $i \leq |\sigma|$ maps variables $a, b, c, \ldots, A, B, C, \ldots$ to data values. Lower-case variables a, b, c,... are called *static* and do not vary over time. Basic ITL contains conventional propositional operators such as $\wedge$ and first-order ones such as $\forall$ and $=$. Normally expressions and formulas are evaluated relative to the beginning of the interval. For example, the formula $J = I + 1$ is true on an interval σ iff the J's value in σ's initial state is one more that I's value in that state.

There are three primitive temporal operators skip, ";" (*chop*) and "*" (*chop-star*). Here is their syntax, assuming that S and T are themselves formulas:

$$\mathsf{skip} \qquad S;T \qquad S^* \ .$$

The formula skip has no operands and is true on an interval iff the interval has length 1 (i. e., exactly two states). Both *chop* and *chop-star* permit evaluation within various subintervals. A formula $S;T$ is true on an interval σ with states $\sigma_0, \ldots, \sigma_{|\sigma|}$ iff the interval can be chopped into two sequential parts sharing a single state σ_k for some $k \leq |\sigma|$ and in which the subformula S is true on the left part $\sigma_0, \ldots, \sigma_k$ and the subformula T is true on the right part $\sigma_k, \ldots, \sigma_{|\sigma|}$. For instance, the formula $\mathsf{skip}; (J = I + 1)$ is true on an interval σ iff σ has at least two states $\sigma_0, \sigma_1, \ldots$ and $J = I + 1$ is true in the second one σ_1. A formula S^* is true on an interval iff the interval can be chopped into zero or more sequential parts and the subformula S is true on each. An empty interval (one having exactly one state) trivially satisfies any formula of the form S^* (including *false**).

Figure 2 pictorially illustrates the semantics of skip, *chop*, and *chop-star*. Some simple ITL formulas together with intervals which satisfy them are shown in Fig. 3.

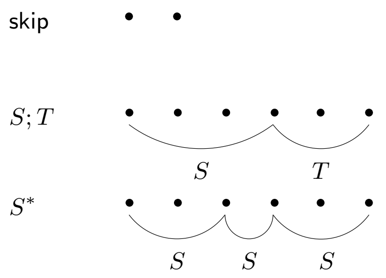

Figure 2. Informal illustration of ITL semantics

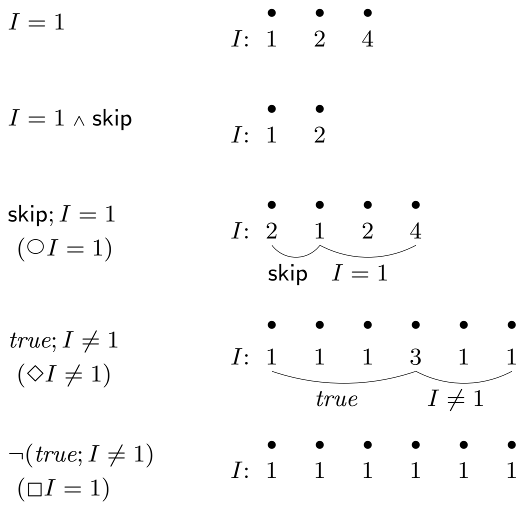

Figure 3. Some sample ITL formulas and satisfying intervals

We generally use w, w', x, x' and so forth to denote *state formulas* with no temporal operators in them. Expressions are denoted by e, e' and so on.

In [41] we make use of the conventional logical notion of *definite descriptions* of the form $\imath v\colon S$ where v is a variable and S is a formula. These allow a uniform semantic and axiomatic treatment in ITL of expressions such as $\bigcirc e$ (e's next value), *fin* e (e's final value) and *len* (the interval's length). For example, $\bigcirc e$ can be defined as follows:

$$\bigcirc e \quad \stackrel{\text{def}}{=} \quad \imath a\colon \bigcirc(e = a) \ ,$$

where a does not occur freely in e. Unit assignment is defined as follows:

$$e := e' \quad \stackrel{\text{def}}{\equiv} \quad \bigcirc e = e' \ .$$

Here is a way to define temporal assignment using a *fin* term:

$$e \leftarrow e' \quad \stackrel{\text{def}}{\equiv} \quad (\mathit{fin}\ e) = e' \ .$$

The operator stable tests whether an expression's value changes:

$$\mathsf{stable}\ e \quad \stackrel{\text{def}}{\equiv} \quad \exists a \cdot \Box(e = a) \ ,$$

where the static variable a is chosen so as not to occur freely in the expression e. The formula $e\ \mathit{gets}\ e'$ is true iff in every unit subinterval, the initial value of the expression e' equals the final value of the expression e:

$$e\ \mathit{gets}\ e' \quad \stackrel{\text{def}}{\equiv} \quad \Box(\mathit{more} \quad \supset \quad e := e') \ .$$

The concrete constructs choice and while are defined as follows:

$$\begin{array}{lcl} \mathsf{if}\ f_0\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2 & \stackrel{\text{def}}{\equiv} & (f_0 \wedge f_1) \vee (\neg f_0 \wedge f_2) \\ \mathsf{while}\ f_0\ \mathsf{do}\ f_1 & \stackrel{\text{def}}{\equiv} & (f_0 \wedge f_1)^* \wedge \mathit{fin}(\neg f_0) \end{array}$$

The parallel construct can simply be expressed as conjunction:

$$f_0 \parallel f_1 \quad \stackrel{\text{def}}{\equiv} \quad f_0 \wedge f_1$$

Figure 4 shows examples of these operators.

		•	•	•	•	•
$\mathsf{stable}\ K$	K:	4	4	4	4	4
		•	•	•	•	•
$K \leftarrow K+1$	K:	2	6	1	8	3
		•	•	•	•	•
$K\ \mathit{gets}\ K+1$	K:	4	5	6	7	8

Figure 4. Sample formulas illustrating *stable*, etc.

4. Verilog Semantics

One of the key strengths of our approach is that both development and its formal verification are uniformly performed within a single logical framework. However, as we are utilizing some existing technologies for hardware synthesis, namely Verilog HDL (see Fig. 1), it is paramount to define a formal semantics for Verilog within the same logical framework. This gives us also the added value of ensuring that the various abstraction gaps during the development, for example between *Tempura-H* and Verilog's behavioral specification (see Fig. 1), are 'soundly' bridged via refinement (see Section 5).

4.1. Verilog syntax

Here we define the syntax of the language we consider. A richer set of Verilog constructs is considered in [15] while here we simplify the language for the sake of brevity. All constructs are given in BNF style description. Because of the specifics of Verilog we consider two syntactical categories namely *statement*, and *atom*.

Table I. Syntax of *statement* and *atom*

$$
\begin{aligned}
\text{statement} &::= \text{empty} \mid \eta \mid \text{block\_assign} \mid \text{if} \mid \text{while} \mid \text{begin\_end}\\
\text{empty} &::= \varepsilon\\
\eta &::= @(e\_exp) \mid \#exp\\
\text{block\_assign} &::= v = exp\\
\text{if} &::= \texttt{if}\ (\texttt{bool})\ statement\ \texttt{else}\ statement\\
\text{while} &::= \texttt{while}\ (\texttt{bool})\ statement\\
\text{begin\_end} &::= \texttt{begin}\ \{statement;\}\ \texttt{end}\\
\\
\text{atom} &::= \texttt{assign}\ v = exp \mid \texttt{always}\ statement \mid \texttt{initial}\ statement
\end{aligned}
$$

A statement is one of the sequential statements of Verilog. These are all statements one may find in a begin_end block for example. All statements are given in table I. There *e_exp* is a boolean expression over event variables, *exp* is an expression, `bool` is a boolean and *event* is an event variable normally declared as `event e`; in a Verilog program. Time delays and event controls are denoted in a standard Verilog manner.

An atom is the smallest unit of parallelism in Verilog. These are the continuous assignment, always and initial constructs. Both Behavioral and RTL language constructs are included. Typically a Verilog program is a collection of atoms with appropriate variable declarations. All atoms run in parallel and share the variables as well as a common clock. A Verilog program is denoted as follows:

$$\begin{aligned}\texttt{program} ::= \ & \texttt{module}\ \text{name}\ (*);\\ & \qquad \textit{global variables};\\ & \qquad atom_1\ ;\ \ldots\ atom_n;\\ & \texttt{endmodule}\end{aligned}$$

4.2. Translating Verilog into Tempura

We define a function that translates Verilog constructs into Tempura equivalents, hence giving semantics for Verilog. The obtained semantics follows a declarative style.

DEFINITION 1. *If* `statement` *is a valid Verilog statement, then* $\|$`statement`$\|$ *gives its Tempura equivalent.*

We would like to introduce here a *naming convention* for all local variables in an atom. Suppose M is an atom and $\mathcal{V}$ is a local variable for M. Then we will write $M.\mathcal{V}$ instead of $\mathcal{V}$ only. For simplicity, we will assume variables in our Tempura specifications unless we explicitly specify the type of the variables in the context.

Suppose we now have a Verilog specification. It defines a set of atoms. Our general idea is to translate all atoms into Tempura formulas and combine them with Tempura's $\wedge$ connective. Assuming the syntax of a Verilog program is as the one given above, the semantics of it will be given as

$$\begin{aligned}&\|\texttt{program}\| ::=\\ &\quad \exists Atom_1.status, \ldots, Atom_n.status,\\ &\quad\ \ global\ variables, Disable, Time \bullet\\ &\quad \{\\ &\qquad global\ variables = \bot \wedge \text{clock}(Disable) \wedge\\ &\qquad \Box(Disable = (Atom_1.status = active \vee \ldots\\ &\qquad\qquad\qquad\qquad \vee\ Atom_1.status = active)) \wedge\\ &\qquad \|atom_1\| \wedge \|atom_2\| \wedge \ldots \wedge \|atom_n\|\\ &\quad \}\end{aligned}$$

where *global variables* is a list of all global for the program and *global variables* $= \bot$ is a shortcut for the initialization to undefined value for all such variables.

4.2.1. *Explicit clock*
We will have an atom called `clock` which will keep the time in a global variable Time. The rationale behind a global clock is that every piece of digital hardware has a clock for synchronization.

$$
\begin{array}{l}
\text{clock}(\mathit{Disable}) \mathrel{\widehat{=}} \mathit{Time} \;=\; 0 \wedge \\
\quad [\\
\qquad \mathsf{while}\,(\mathit{Disable})\ \mathsf{do\ skip}; \\
\qquad \mathit{Time} := \mathit{Time} + 1\,;\mathsf{skip} \\
\quad]^{*}
\end{array}
$$

The clock in our specification has one parameter namely the state variable *Disable* which synchronizes all atoms. When *Disable* is *true* then the clock is simply doing nothing. Once all atoms are suspended, then *Disable* turns into *false* and the clock advances the time.

The full semantics for all atoms is given in [14] so we will only mention the semantics of `assign` for illustrative purposes.

4.2.2. *Assign*
As shown in table I, the form of the `assign` statement is `assign` $v = exp(v_1, \ldots, v_n)$. The Tempura equivalent for it has one free variable, i.e. $M.status$, which synchronizes its atom with the global clock. $\perp$ denotes undefined value.

$$
\begin{array}{l}
\|\mathtt{assign}\ v = exp(v_1, \ldots, v_n)\| ::= \\
\quad \exists M.v_1, \ldots, M.v_n \bullet M.status = active \wedge \\
\qquad M.v_1 = \perp \wedge \ldots \wedge M.v_n = \perp \wedge v = \perp \wedge \\
\qquad [\ \mathsf{if}\,(v_1 = M.v_1, \ldots, v_n = M.v_n)\ \mathsf{then}\ M.status := suspend \\
\qquad\ \ \mathsf{else}\ (\\
\qquad\quad v := exp(v_1, \ldots, v_n) \wedge M.status := active \wedge \\
\qquad\quad M.v_1 := v_1 \wedge \ldots \wedge M.v_n := v_n \\
\qquad\ \ ) \\
\qquad]^{*}
\end{array}
$$

Informally, we can see here an indefinite loop defined by $[\ldots]^*$ in which we check if the assignment is scheduled for the current time instance or the triggering variables have changed. If neither of these occurs, we simply suspend the assign atom. Otherwise, depending on the condition, we either activate the assignment or we reschedule it for a later time instance.

5. Formal Refinement and Analysis

5.1. Refinement Calculus

Each step of the design process (see Fig 1) involves the application of a single refinement rule in our calculus. The rules are applied automatically (within the HOL system) but the choice of which rule to apply at each step is not automated. So far, we have implemented a basic library of refinement rules. These are very general but represent too small steps for a real system and we envisage that these rules would be combined into higher-level and application-specific design steps, perhaps encompassing whole design strategies. More experience is needed to implement such higher-level rules.

The refinement relation $\sqsubseteq$ is defined as follows: A system Sys is *refined* by a more concrete system Sys', denoted $Sys \sqsubseteq Sys'$, if and only if $Sys' \supset Sys$. A set of sound refinement laws have been derived [8] to transform an abstract system specification into concrete systems. Furthermore a number of refinement rules were developed in HOL [25].

Two observations are in order:

1. Once we have completed the formal specification phase, various properties could be proven about the specification itself. This can provide an extra assurance that the final specification meets the required informal requirements.
2. At each refinement step, we can simulate the resulting (sub)system. This gives some guidelines on the choice of the subsequent refinement rules.

We have implemented rules that are applicable in both control- and data-dominated applications, though the examples in this paper concentrate on functional correctness. For an example of timing-specific refinement, see for example [8].

The following are some useful refinement rules for refining ITL specifications into Tempura code. This set of rules is by no means complete but gives just a flavor of the type of rules. The conditional is introduced with the following rule.

RULE 1 (If then else).

$$(\mathsf{if}-1)\ \ (f_0 \wedge f_1) \vee (\neg f_0 \wedge f_2) \quad \sqsubseteq \quad \mathsf{if}\ f_0\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2$$

Chop has empty as a unit, is associative and distributes over nondeterministic choice and conditional

RULE 2 (Chop).

$$
\begin{array}{llcl}
(;-1) & \mathsf{empty} \mathbin{;} f \equiv f & \equiv & f \mathbin{;} \mathsf{empty} \\
(;-2) & (f_1 \mathbin{;} f_2) \mathbin{;} f_3 & \equiv & f_1 \mathbin{;} (f_2 \mathbin{;} f_3) \\
(;-3) & f_1 \mathbin{;} (f_2 \vee f_3) \mathbin{;} f_4 & \equiv & (f_1 \mathbin{;} f_2 \mathbin{;} f_4) \vee (f_1 \mathbin{;} f_3 \mathbin{;} f_4) \\
(;-4) & (\mathsf{if}\ f_0\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2) \mathbin{;} f_3 & \equiv & \mathsf{if}\ f_0\ \mathsf{then}\ (f_1 \mathbin{;} f_3)\ \mathsf{else}\ (f_2 \mathbin{;} f_3)
\end{array}
$$

The following rules introduce the while loop and the non-terminating loop

RULE 3 (While).

$$
\begin{array}{llcl}
(\mathsf{while}\ -1) & (f_0 \wedge f_1)^* \wedge \mathit{fin}\ \neg f_0 & \sqsubseteq & \mathsf{while}\ f_0\ \mathsf{do}\ f_1 \\
(\mathsf{while}\ -2) & f_1^* & \sqsubseteq & \mathsf{while}\ \mathit{true}\ \mathsf{do}\ f_1
\end{array}
$$

The following rules are used in JPEG example in Section 6.2. The 'Split assignment' rule splits an assignment into two sequentially composed assignments.

RULE 4 (Split assignment).

$$Z \leftarrow g(f(X)) \quad \sqsubseteq \quad \exists Y \bullet Y \leftarrow f(X) \mathbin{;} Z \leftarrow g(Y)$$

The following rule, where $X \leftarrow_V e$ denotes an assignment with program variables V permits replacement of concurrent assignments.[1]

RULE 5 (Split concurrent assignment).

$$X_1, X_2 \leftarrow_V e_1, e_2 \ \sqsubseteq\ X_1 \leftarrow_V e_1 \mathbin{;} X_2 \leftarrow_V e_2$$

provided that X_1 and X_2 differ, X_1 is not free in e_2 and V contains X_1 and the free variables in e_2. A generalized version of this rule is mechanized in the tools integrated into the ITL workbench. The following rule is used to introduce implementation details.

RULE 6 (Implementation details intro).

$$f_0 \sqsubseteq f_0 \wedge f_1$$

The following Verilog rules are used in the sorter application in Section 6.1. They enable the transformation of Tempura constructs into Verilog constructs.

[1] The program variables in V are stable (i.e. retain their values) if not explicitly assigned, as in conventional programming languages, but unlike logical variables.

RULE 7 (Verilog).

$$
\begin{array}{ll}
(Verilog-1) & \mathsf{while}\ true\ \mathsf{do}\ f(X,Y) \\
 & \sqsubseteq \\
 & \text{“}always@(X)\ f;\text{”} \\
(Verilog-2) & \exists x \bullet \{x = Clock \wedge \mathsf{while}\ Clock < x+T\ \mathsf{do\ skip}\}\ ;\ f \\
 & \sqsubseteq \\
 & \text{“}\#T\ f;\text{”} \\
(Verilog-3) & A = e_0\ \wedge B = e_1 \\
 & \sqsubseteq \\
 & \text{“}A <= e_0;\ B <= e_1;\text{”}
\end{array}
$$

where X is only read by f, and $Clock$ is a global clock, and A and B are not the same variable. Note: we have used the ".." to indicate a Verilog construct to avoid confusion because Verilog uses for example "always" and ";" but these have a different meaning than the ITL/Tempura constructs with the same name.

5.2. Runtime Analysis

A fundamental characteristic of our approach is the ability to capture a possible partial behavior of a running (sub-)system. Once the behavior is captured then we can assert if such behavior satisfies a given property, i.e., runtime validation. We are not dealing here with the formal verification of properties which requires that all possible behaviors of system satisfy the properties but we are rather concerned with validating properties which requires that only interesting behaviors satisfy the properties.

The importance of the AnaTempura tool in this context has two aspects

– it integrates simulation in our framework

– we can use the approximate behavior acquired for selecting the right refinement rule to be applied next [48].

The states of a (sub-)system to be analyzed are captured by inserting *assertion points* at suitably chosen places. These divide the system into several *code-chunks*. Properties of interests are then validated for this behavior.

Our general framework for analysis can be described as follows.

1. Establish all desirable properties of the system under consideration and express them in Tempura.
2. Identify suitable places in the code and insert assertional points.

3. Using Tempura, check that the behavior satisfies the desired properties.

Obviously, some level of understanding of the (sub-)system under consideration is assumed. These properties could be invariants that need to be true at all levels of system's abstraction.

The locations of assertion points could be chosen, for example, at the entry and exit points of a procedure or function. In this case assertions are in fact *pre-* and *post-* conditions, and what we are asserting is: If the system starts at a state satisfying the *pre-* condition then it terminates properly in a state satisfying the *post-* condition.

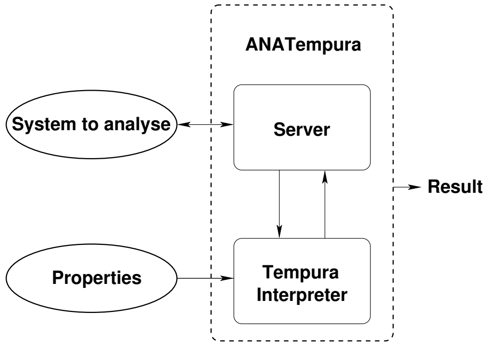

Figure 5. Basic Functions

We have designed and implemented a tool, known as AnaTempura [48], that support the approach described above. This was integrated within the ITL Workbench. Figure 5 shows the general structure of the tool. The inputs are the system description (either source code plus assertion points or an ITL specification) and the properties we want to check. The result of the analysis is whether the properties hold for the system. Optionally the behavior of the system can be animated. Currently the tool can analyze C, Verilog and Tempura programs. The tool is available from [32].

6. Application

In this section we give two applications to illustrate our approach. The first is a JPEG encoder and the second is a sorter system to demonstrate migration policies of legacy software applications.

The sorter example will show how to migrate part of a C program into a Verilog description using the refinement rules of Section 5. It also shows how simulation can help in the migration process.

Although the JPEG encoder is a stock item nowadays and therefore not a real-world design challenge, it has become something of a benchmark example in the co-design literature. In this section, we look at it from the point of view of the ITL Workbench, as an illustration of the application of formally-based design. We show how to derive a synthesisable implementation completely formally in a way that preserves functional correctness and is uniform across all levels of design. At the higher levels of design, the choice of refinement rules is a matter of judgment. When we arrive at a concrete hardware description, we can synthesize a Tempura automaton automatically. We do not address issues of timing or optimal partitioning in this section, although these are of course very important parts of the overall design and may be addressed in our framework.

6.1. Legacy Migration

This section discusses the migration of part of a software system into hardware. The system that we want to migrate is used to sort first and second class letters into trays. The system is depicted in Fig. 6.

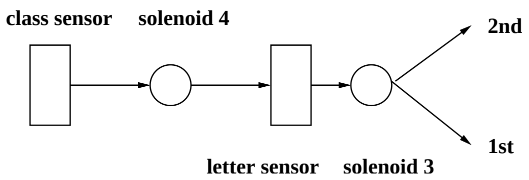

Figure 6. Letter sorter

The sorter consists of 2 sensors for detecting respectively the class of a letter and whether a new letter has arrived. Furthermore it has 2 solenoids for respectively holding up a letter temporarily (solenoid 4) and switching the direction of the tray (solenoid 3). A fragment of the C code is shown below.

```
scan_csensor (&class_sensor);
 if (class_sensor < 2)
   {
     assertion("class", 1);
     SolOff(4); Delay(delay4,1); SolOn(4);  Delay(delayF,4);
     scan_lsensor (&letter_sensor);
     assertion("lsens",letter_sensor);
      if ( !YellowSet )
       { Delay(delay3A,2); SolOff(3); Delay(delay3B,3); YellowSet = 1; }
   }
 else
   {
     assertion("class",2);
     SolOff(4); Delay(delay4,1); SolOn(4); Delay(delayF,4);
```

```
    scan_lsensor (&letter_sensor);
    assertion("lsens",letter_sensor);
    if ( YellowSet )
      { Delay(delay3A,2); SolOn(3); Delay(delay3B,2); YellowSet = 0; }
  }
```

The delays in the code are crucial in that they ensure that once a first class letter has been detected it ends up in the first class tray. The problem occurs when *migrating* the software to a new hardware platform. As the delays are implemented in software using counters and the speed of the new machine is different from the old one, our software solution becomes invalid.

A test and change cycle was adopted to the delay till the sorter worked again. The results of these change were also used to implement the delays in hardware to avoid having the problem in the future. For this test/change and the hardware implementation we inserted assertion points in the code.

AnaTempura was used to check properties of the sorter, i.e., what are the correct values of the delays in order for the sorter to work correctly. A screen dump of the result is shown in Fig 7.

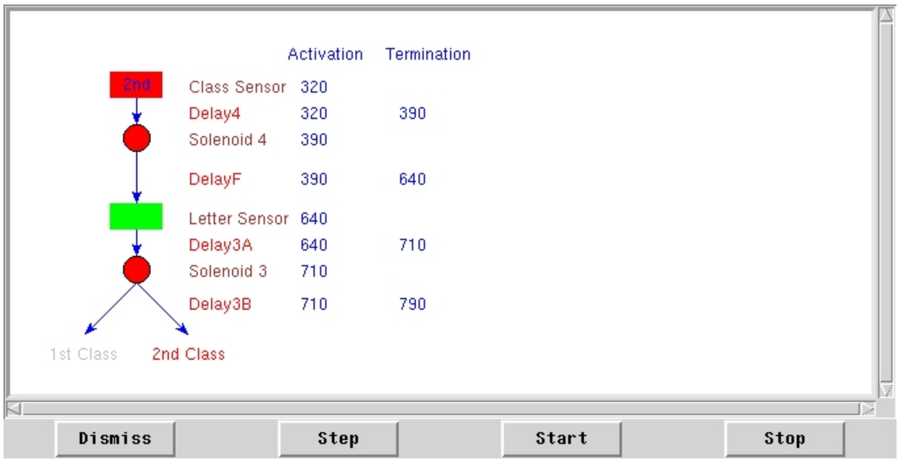

Figure 7. Screen dump

According to our formal framework, we first have to give an Tempura (ITL) specification of our delay construct. The following is such a specification where `Clock` is our global clock.

```
define delay(T) =
        { exists x : { x=Clock and while Clock < x+T do skip } }
```

Note: We could have started from the following ITL specification:

$$\mathsf{delay}(T) \doteq \exists x \cdot x = Clock \wedge \mathit{fin}\ Clock = x + T$$

and use the refinement rules to derive above Tempura code.

The following program will create a test run of this delay construct.

```
 define main() = {
   define text(X) = { empty  and  format("%s\n",X)}
   and
   exists Clock : {
   define delay(T) = {
      exists x: {
        x=Clock and  while Clock < x+T  do skip
     }
  }
   and
  Clock = 0  and Clock  gets Clock + 1  and
   always (  format("Clock=%t \n",Clock) )  and
  {  skip;
    text("start delay");
    delay(5);
    text("end delay");
     skip; skip;
    text("start delay");
    delay(10);
    text("end delay");
     skip
  }
 }
}.
```

Using AnaTempura to execute above program we get the following output:

```
Tempura 4>
State   0: Clock=0
State   1: Clock=1
State   1: start delay
State   2: Clock=2
State   3: Clock=3
State   4: Clock=4
State   5: Clock=5
State   6: Clock=6
State   6: end delay
State   7: Clock=7
State   8: Clock=8
State   8: start delay
State   9: Clock=9
State  10: Clock=10
State  11: Clock=11
State  12: Clock=12
State  13: Clock=13
State  14: Clock=14
State  15: Clock=15
State  16: Clock=16
State  17: Clock=17
State  18: Clock=18
State  18: end delay
State  19: Clock=19
```

```
Done!  Computation length:  19.  Total Passes:  21.
Total reductions:  514  (510 successful).
Maximum reduction depth:  12.
Tempura 5>
```

We will now refine this Tempura specification into Verilog. First we must determine how the software system "communicates" with the hardware, i.e., we have to determine the interface.

- at hardware level: *delay*($T, Setdelay, Enddelay$) where
  - input T indicates how long the delay should be.
  - input $Setdelay$, if set the delay will start.
  - output $Enddelay$, if set the end of the required delay has been reached.
- at software level we have two functions:
  1. *start_delay*($Tdelay$): this will start the delay with a value of $Tdelay$ units.
  2. *wait*(): this will wait until the delay has elapsed.

This means that the software and hardware run in parallel (true concurrency) but the Tempura specification was such that software and hardware run in interleaving mode. So this means we must have another Tempura implementation. Using the refinement rules of Sect. 5 we derive the following Tempura implementation:

```
define delay(T,Setdelay,Enddelay) = {
     if Setdelay=1  then {
      exists x : {
          x=Clock  and Enddelay=0  and
          { while Clock < x+T  do { skip  and Enddelay=0};{ skip  and Enddelay=1}}
         }
    }  else { Enddelay=0  and  skip }
}
```

The test program changes accordingly, i.e., the hardware runs now in parallel and it has the above discussed interface between software and hardware:

```
define main() = {
  exists Clock,T,Setdelay,Enddelay : {
  define text(X) = { empty  and  format("%s\n",X)}  and
  define delay(T,Setdelay,Enddelay) = {
     if Setdelay=1  then {
     exists x: {
      x=Clock  and Enddelay=0  and
      { while Clock < x+T  do { skip  and Enddelay=0};{ skip  and Enddelay=1}}
    }
   }  else { Enddelay=0  and  skip }
```

```
 }
  and
  define start_delay(Tdelay) = { T=Tdelay  and Setdelay=1  and  empty }  and
  define wait() = {  while Enddelay=0  do {
                           skip  and  stable T  and  stable Setdelay} }  and
 Clock = 0  and Clock  gets Clock + 1  and
  always (  format("Clock=%t T=%t Setdelay=%t %Enddelay=%t\n",
                           Clock,T,Setdelay,Enddelay) )  and
  while  true  do delay(T,Setdelay,Enddelay)  and
 { { skip  and T=0  and Setdelay=0};
   text("start delay");
   start_delay(5);
   wait();
   text("end delay");
    skip;{ skip  and T=0  and Setdelay=0};
   text("start delay");
   start_delay(10);
   wait();
   text("end delay");
    skip;{ skip  and T=0  and Setdelay=0};
 }
}
}.
```

Using AnaTempura to execute above test program we get the following output:

```
Tempura 7>
State   0: Clock=0 T=0 Setdelay=0 Enddelay=0
State   1: start delay
State   1: Clock=1 T=5 Setdelay=1 Enddelay=0
State   2: Clock=2 T=5 Setdelay=1 Enddelay=0
State   3: Clock=3 T=5 Setdelay=1 Enddelay=0
State   4: Clock=4 T=5 Setdelay=1 Enddelay=0
State   5: Clock=5 T=5 Setdelay=1 Enddelay=0
State   6: Clock=6 T=5 Setdelay=1 Enddelay=1
State   6: end delay
State   7: Clock=7 T=0 Setdelay=0 Enddelay=0
State   8: start delay
State   8: Clock=8 T=10 Setdelay=1 Enddelay=0
State   9: Clock=9 T=10 Setdelay=1 Enddelay=0
State  10: Clock=10 T=10 Setdelay=1 Enddelay=0
State  11: Clock=11 T=10 Setdelay=1 Enddelay=0
State  12: Clock=12 T=10 Setdelay=1 Enddelay=0
State  13: Clock=13 T=10 Setdelay=1 Enddelay=0
State  14: Clock=14 T=10 Setdelay=1 Enddelay=0
State  15: Clock=15 T=10 Setdelay=1 Enddelay=0
State  16: Clock=16 T=10 Setdelay=1 Enddelay=0
State  17: Clock=17 T=10 Setdelay=1 Enddelay=0
State  18: Clock=18 T=10 Setdelay=1 Enddelay=1
State  18: end delay
State  19: Clock=19 T=0 Setdelay=0 Enddelay=0
...
```

Next step is to transform the Tempura description into a Verilog description. We will use the Verilog refinement rules $Verilog$-1,2,3. The resulting Verilog description is as follows:

```
module delay(T,Setdelay,Enddelay);
   input [7:0] T;
   input       Setdelay;
   output      Enddelay;
   reg         Enddelay;

   always @(T or Setdelay)
     begin
        if (Setdelay==1)
          begin
             if (T==0) Enddelay=1;
             else  begin Enddelay=0; #T Enddelay=1;  end
          end
        else Enddelay=0;
     end
endmodule
```

We also use the refinement rules to translate the Tempura test program into a Verilog test program:

```
module test;
   reg [7:0] T;
   reg       Setdelay;
  delay  delayhw(T,Setdelay,Enddelay);
   initial
     begin
       T<=0;Setdelay<=0;
       #1 T<=5; Setdelay<=1;
        while (Enddelay==0) #1;
       T<=0; Setdelay<=0;
       #1 T<=10; Setdelay<=1;
        while (Enddelay==0) #1;
       T<=0; Setdelay<=0;
       #2 $finish;
     end
  always
   #1 $display($time-1,"    T=%d Setdelay=%d Enddelay=%d",T,Setdelay,Enddelay);
endmodule
```

Using a Verilog simulator this will generate the following output:

```
 0    T=  0 Setdelay=0 Enddelay=0
 1    T=  5 Setdelay=1 Enddelay=0
 2    T=  5 Setdelay=1 Enddelay=0
 3    T=  5 Setdelay=1 Enddelay=0
 4    T=  5 Setdelay=1 Enddelay=0
 5    T=  5 Setdelay=1 Enddelay=0
 6    T=  5 Setdelay=1 Enddelay=1
 7    T=  0 Setdelay=0 Enddelay=0
 8    T= 10 Setdelay=1 Enddelay=0
 9    T= 10 Setdelay=1 Enddelay=0
10    T= 10 Setdelay=1 Enddelay=0
11    T= 10 Setdelay=1 Enddelay=0
12    T= 10 Setdelay=1 Enddelay=0
13    T= 10 Setdelay=1 Enddelay=0
14    T= 10 Setdelay=1 Enddelay=0
15    T= 10 Setdelay=1 Enddelay=0
```

```
                 16     T= 10 Setdelay=1 Enddelay=0
                 17     T= 10 Setdelay=1 Enddelay=0
                 18     T= 10 Setdelay=1 Enddelay=1
                 19     T=  0 Setdelay=0 Enddelay=0
Exiting VeriWell for SPARC at time 21
0 Errors, 0 Warnings, Memory Used: 39282
Compile time = 0.0, Load time = 0.0, Simulation time = 0.0
```

6.2. JPEG Encoder

JPEG is a standard method for image compression based on the discrete cosine transform (DCT). The key feature of the DCT is that it produces many small or zero valued entries in the transformed data which are subsequently quantized (to 0) and compressed by run-length encoding. The structure of a JPEG encoder[2] is illustrated in Figure 8.

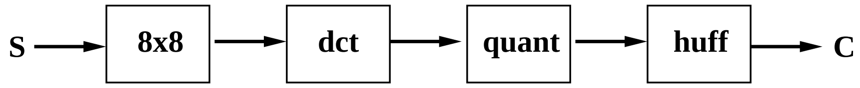

Figure 8. JPEG Encoding

There are four components, repeatedly executed in sequence on the original $m \times n$ image. The image is broken into 8×8 blocks, then each block is passed through the DCT, quantized and finally a compressed using Huffman encoding. The corresponding specification for an $m \times n$ image is given in the following ITL/Tempura formula:

$$
\begin{array}{lll}
jpeg(S,C) & \widehat{=} & \exists X, Y, Z \cdot \\
& & \textsf{for } i < (n+7)/8 \textsf{ do} \\
& & \quad \textsf{for } j < (m+7)/8 \textsf{ do } \{ \\
& & \quad\quad \forall k < 8, l < 8 \cdot X_{k,l} \leftarrow S_{max(i+k,n),max(j+l,m)}; \\
& & \quad\quad dct(X,Y); \\
& & \quad\quad quant(Y,Z); \\
& & \quad\quad huff(Z,C) \\
& & \quad \}
\end{array}
$$

The most computationally intensive part of the JPEG algorithm is the DCT and we focus on that below. The quantization component rounds the transformed data to integer values

$$quant(Y,Z) \ \widehat{=}\ \forall i < 8, j < 8 \cdot Z_{i,j} \leftarrow round(Y_{i,j}/q_{i,j})$$

for given quantization table q, reducing many values to zero. This loses information but aids compression. The Huffman encoding treats the so-called DC coefficient, $Z_{0,0}$, separately from the other 63, so-called AC,

[2] Strictly speaking, we consider only sequential DCT-mode, which is the most common.

coefficients. The main reason is that the DC coefficient is a measure of the average value of the image samples and DC coefficients are likely to be closely related between adjacent image blocks, permitting a more efficient encoding. Also, the AC coefficients are encoded in a "zigzag" order, $Z_{0,1}, Z_{1,0}, Z_{2,0}, Z_{1,2}, Z_{0,2}, Z_{0,3}, \ldots$, because those near the beginning of this order are the most significant and those further down the order are more likely to be zero, which improves compression.

$$\begin{aligned} huff(Z, C) \;\widehat{=}\; \exists zz \cdot\; & zigzag(Z, zz); \\ & encodeDC(zz_0, C); \\ & encodeAC(zz, C) \end{aligned}$$

Our method involves mechanical refinement of this specification into an implementation using tools available in the ITL workbench (see Fig. 1). We will develop two different designs for the DCT component: a) parallel and b) pipelined and describe their integration into the complete design.

6.2.1. *DCT Refinement: parallel*

The steps involved in the refinement of our DCT component are described below. These steps were performed in our refinement tool, based on HOL's Window Inference package [21].

The standard definition of the DCT is expressed in ITL as follows:

$$dct(X, Y) \;\widehat{=}\; \forall i, j \cdot Y_{i,j} \leftarrow \frac{\alpha(i)\alpha(j)}{4} \sum_{m=0}^{7} \sum_{n=0}^{7} X_{m,n}\gamma(i, m)\gamma(j, n)$$

where

$$\begin{aligned} \alpha(i) \;&\widehat{=}\; \mathsf{if}\,(i = 0)\;\mathsf{then}\;\sqrt{\frac{1}{2}}\;\mathsf{else}\;1 \\ \gamma(i, j) \;&\widehat{=}\; cos\frac{(2j + 1)\pi i}{16} \end{aligned}$$

This is computationally inefficient ($O(N^4)$) and can be improved using standard row-column decomposition techniques to the following $O(2N^3)$ algorithm:

$$dct(X, Y) \;\sqsubseteq\; Y \leftarrow AXA^T$$

where $A_{m,n} = \frac{\alpha(m)\gamma(m,n)}{2}$. This refinement can most easily be performed in a single step by postulating the result and proving the corresponding refinement rule by expansion of the formulae.

The refined specification $Y \leftarrow AXA^T$ is further decomposed into two identical steps using the sequential refinement rule 4 and the fact that $A(AX^T)^T = AXA^T$, giving

$$dct(X,Y) \sqsubseteq \exists Z \bullet Z \leftarrow AX^T \,;\, Y \leftarrow AZ^T$$

Each part of the sequential composition is then decomposed into parallel 1-D DCTs using the theorem $Y \leftarrow AX^T \sqsubseteq \forall i < 8 \bullet dct1(X_i, Y_i)$, where

$$dct1(X,Y) \;\widehat{=}\; Y \leftarrow AX^T$$

to give

$$dct(X,Y) \sqsubseteq \exists Z \bullet (\forall i < 8 \bullet dct1(X_i, Z_i)) \,;\, (\forall i < 8 \bullet dct1(Z_i^T, Y_i))$$

The resulting architecture is shown in figure 9.

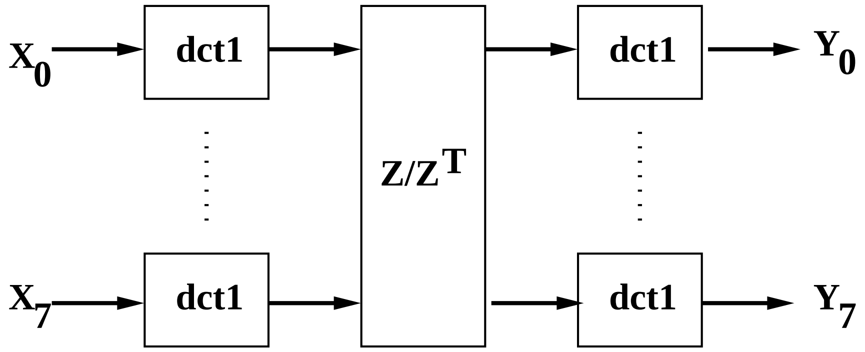

Figure 9. Parallel 1-D DCT Architecture

The 1-D DCT can be refined in many ways and there are numerous published algorithms. Generally, the aim is to reduce the number of multiplications by exploiting symmetries of the cosine function. Here, we choose an algorithm due to Chen [10] which reduces the problem to two 4×4 matrix multiplications and requires 4 steps with a total of 16 multiplications. The algorithm is captured in the ITL specification below

$$
\begin{aligned}
chen(X,Y) \;\widehat{=}\;\; & Y_0, Y_1, Y_2, Y_3, Y_4, Y_5, Y_6, Y_7 := \\
& \quad (X_0 + X_7)/2, (X_1 + X_6)/2, (X_2 + X_5)/2, \\
& \quad (X_3 + X_4)/2, (X_3 - X_4)/2, (X_2 - X_5)/2, \\
& \quad (X_1 - X_6)/2, (X_0 - X_7)/2; \\
& Y_0, Y_1, Y_2, Y_3, Y_4, Y_5, Y_6, Y_7 := \\
& \quad Y_0 + Y_3, Y_1 + Y_2, Y_1 - Y_2, Y_0 - Y_3, \\
& \quad Y_4, d.(Y_6 - Y_5), d.(Y_6 + Y_5), Y_7; \\
& Y_0, Y_1, Y_2, Y_3, Y_4, Y_5, Y_6, Y_7 := \\
& \quad d.(Y_0 + Y_1), d.(Y_0 - Y_1), f.Y_2 + b.Y_3, f.Y_3 - b.Y_2, \\
& \quad Y_4 + Y_5, Y_4 - Y_5, Y_7 - Y_6, Y_7 + Y_6; \\
& Y_0, Y_4, Y_2, Y_6, Y_1, Y_5, Y_3, Y_7 := \\
& \quad Y_0, Y_1, Y_2, Y_3, \\
& \quad g.Y_4 + a.Y_7, c.Y_5 + e.Y_6, c.Y_6 - e.Y_5, g.Y_7 - a.Y_4
\end{aligned}
$$

where a, b, c, d, e, f and g are cosine coefficients.

It is not really feasible to derive an algorithm like Chen's by pure top-down refinement. Instead, we find the algorithm by other means and verify the refinement step, $dct1(X,Y) \sqsubseteq chen(X,Y)$, mechanically by expansion. The architecture specification $chen(X,Y)$ is also executable and the development of such an algorithm can be assisted by testing and simulation in the Tempura system before attempting the refinement.

6.2.2. *Partitioning*

Once the architecture has been refined, the components must be allocated to hardware and software based on conventional methods, such as co-simulation. The method used is not an issue here and does not impact on the correctness of the final implementation (though may affect it's performance). The formal refinement steps we use are guaranteed to preserve correctness. In the JPEG example the DCT is the most computationally demanding part and is therefore the function most naturally assigned to hardware (see for example [2]).

We are targeting a co-processor implementation, with the DCT implemented on special-purpose hardware and the rest of the system in software. The hardware and software parts of the system communicate asynchronously via a bus, so communication timing is not an issue.

In order to refine the communication between the software part and the DCT co-processor, we introduce send and receive predicates for bus communication. These predicates have the following properties.

$$
\begin{aligned}
Y \leftarrow X \;&\sqsubseteq\; snd(B,X) \wedge rcv(B,Y) \\
stb(B) \,;\, rcv(B,X) \;&\sqsubseteq\; rcv(B,X)
\end{aligned}
$$

Based on these primitives, we obtain a *co-processor refinement rule*,

$$\begin{aligned} Y \leftarrow f(X) \;\sqsubseteq\; & (snd(B,X)\,;rcv(B,Y)) \wedge \\ & (\exists Z \bullet rcv(B,Z)\,;Z \leftarrow f(Z)\,;snd(B,Z)) \end{aligned}$$

under suitable conditions on f. The same rule will hold for many different bus designs. The refined specification has now become

$$dct1(X,Y) \;\sqsubseteq\; \exists B \bullet bus(B) \wedge sw(B,X,Y) \wedge hw(B)$$

where hw and sw refer to the hardware and software parts of the DCT.

$$\begin{aligned} sw(B,X,Y) \;&\widehat{=}\; snd(B,X)\,;rcv(B,Y) \\ hw(B) \;&\widehat{=}\; \exists Z \bullet rcv(B,Z)\,;chen(Z,Z)\,;snd(B,Z) \end{aligned}$$

The refined architecture is illustrated in Figure 10.

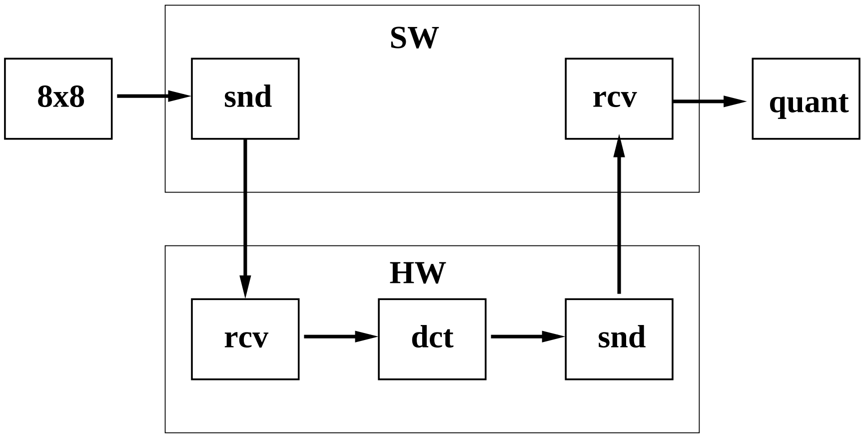

Figure 10. Co-processor Architecture

The software parts of the system are refined into sequential Tempura. This is achieved by mapping parallel to sequential structures. For example, restricted universal quantifiers are transformed to loops, existential quantifiers are transformed to local variable declarations, concurrent assignments are transformed to sequential.

Using these transformations together with some facts about rounding, the quantization component may be refined to the following sequential Tempura:

$$\begin{aligned} quant(Y,Z) \;\sqsubseteq\; & \textsf{for } i < 8, j < 8 \\ & \quad \textsf{if } 0 < Y_{i,j} \\ & \quad\quad \textsf{then } Z_{i,j} := (Y_{i,j} + q_{i,j}/2)/q_{i,j} \\ & \quad\quad \textsf{else } Z_{i,j} := (Y_{i,j} - q_{i,j}/2)/q_{i,j} \end{aligned}$$

6.2.3. *Implementation*

Tempura is not an implementation language but may be converted into an implementation. The sequential part of Tempura is already very close to a conventional sequential language and may readily be translated, however Tempura can also be automatically refined to an automaton from which an implementation can be synthesized. The idea is to translate a Tempura program p (in suitable form) to an automaton $\mathcal{A}(p, Rst, Rdy)$ such that

$$p \sqsubseteq (Rst \wedge \textsf{while}\,(\neg Rdy)\ \textsf{do}\ \mathcal{A}(p, Rst, Rdy))$$

The program starts on the Rst signal, then the loop executes one state transition of the automaton per cycle until the Rdy flag is set, indicating termination (Rdy is equivalent to $\textsf{empty}$). The translation is defined recursively over Tempura constructs and the resulting automaton is a Tempura program that can be simulated.

The refined DCT hardware block has been automatically refined to a Tempura automaton in this way. This can then be refined to Verilog (or Handel) and synthesized to produce an FPGA implementation.

6.2.4. *Pipelined systolic array implementation*

We use transformational methods for implementing algorithms in hardware which provide alternative implementation schemes. Two such methods are:

1. Regular array synthesis — deriving regular array architectures from high-level specifications [35, 39]. In this scheme concurrency is implicit and the design process allows us to systematically introduce concurrency while taking into account the geometry of interconnection of the designs. This method can deliver architectures with local interconnection pattern. Although only a certain class of interconnection patterns can be realized in this method the advantages are a) automatic derivation of pipelined design b) formal basis, i.e., designs are correct by construction.

2. Hardware compilation — implementing concurrent programs as hardware circuits [46, 43]. In this method the concurrency is made explicit through programming language features and this provides a degree of freedom to explicitly control the communication patterns of a design and therefore explore a wider design space. However, the correctness criteria has to be established explicitly.

Regularity in computations is a feature of the DCT algorithm and it is for this reason that we will be able to apply regular array synthesis techniques to design suitable hardware components with local interconnection. These transformations when applied to an initial high-level specification can progressively refined it into a specification for a regular array. The array description can then be directly translated into formats for hardware synthesis. One possible route to implementation is through the Handel compilation scheme as the Handel language provides the appropriate abstraction for describing computations in a synchronous array of processors and pipelining of data over synchronized channels.

Let us consider the derivation of a pipelined architecture for the DCT. The steps involved in the refinement are as follows. We start with the $O(N^3)$ algorithm:

$$dct(X,Y) \sqsubseteq Y \leftarrow AXA^T$$

and decompose the specification $Y \leftarrow AXA^T$ into two components:

$$dct1(X,G) \;\widehat{=}\; G \leftarrow XA^T \quad \text{and} \quad dct2(G,Y) \;\widehat{=}\; Y \leftarrow AG$$

so that regular array designs for each multiplication can be developed separately and the separately evolved designs are composed to obtain the global regular array design with the desired characteristics by a suitable composition scheme (ψ):

$$\begin{aligned} array(X,Y) \;&\widehat{=}\; \psi(array1, array2) \\ dct1 \;&\sqsubseteq\; array1 \\ dct2 \;&\sqsubseteq\; array2 \end{aligned}$$

The *dct*1 component is refined into a regular array using the following refinement steps

Specification: In the specification step we convert a mathematical specification into a system of recurrence equations (SRE) — a specification format for analysis and synthesis of regular arrays [35]. An SRE for multiplying two matrices can be directly expressed as follows:

$$\begin{aligned} dct_1^a \;&\widehat{=}\; ((\forall i,j \bullet 1 \le i,j \le N)\; Gred(X, A^T)) \\ Gred(X, A^T) \;&\widehat{=}\; G[i,j] = reduce(+, (i,j,k \to k), X[i,k] * A^T[k,j]) \end{aligned}$$

where the *reduce* operation corresponds to the reduction operation (summation of partial products) in a matrix multiplication.

Regularisation: dct_1^a is next refined into a System of Uniform Recurrence Equations (SURE) [38] dct_1^b where

$$dct_1^b \;\widehat{=}\; \exists G, X1, A1 \bullet reduce(G, X1, A1) \wedge pipe(X1, X) \wedge pipe(A1, A)$$

Although dct_1^b is depicted as concurrent evaluation of variables G, X and A, the evaluation is not fully parallel but is constrained by the *dependences* that are defined by the equations.

The order of evaluation (schedule) and the structure of the processor array in which the computations are carried out (allocation) are derived by *space-time* transformation [35].

Space-time transformation: The spatio-temporal attributes of a design are defined in this step. The particular form of this transformation determines the *pipelined* nature of execution. The transformation is carried out by computing a transformation function $\mathcal{T}$ which is a matrix of dimension 3×3 and applying it to dct_1^b. This then gives a specification of a regular array $array1$ which is a refinement of dct_1^b. In this array the inputs are pipelined at the boundary processors and each processor computes and stores one element of the result matrix G_{ij}. The important aspect of the design to note is that all communications are between neighboring processors and therefore this architecture avoids global communications.

For the second multiplication $dct2$ we can (re-)use the dct_1^b derived above. Details of this transformation can be found in [35, 37].

We now compose these two designs to obtain the global design for the DCT using composition scheme ψ. The effect of this composition is to produce a *composite* array in which the computations of dct_1 and dct_2 are pipelined. By cascading the result in this pipelined fashion the architecture not only avoids transposition that are needed in other DCT designs but also improves upon the latency of the design.

The architectural specification for each multiplication can be translated into Handel programs and therefore can be synthesized into hardware circuits. This final bridging with hardware synthesis is currently not automatic although it is feasible to bridge the gap between design and synthesis tools.

The interaction between hardware and software part is structurally equivalent to the one described in section 6.2.2. We can continue to retain the bus strategy for communicating between the two parts and hence the co-processor refinement rule remains unaltered.

References

1. Allara, A., C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto: 1998, 'System-Level Performance Estimation Strategy for Sw and Hw'. In: *Proceedings of the International Conference on Computer Design 1998*.
2. Balarin, F., M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara: 1997, *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Publishers.
3. Barringer, H., M. Fisher, G. Gough, D. Gabbay, and R.Owens: 1995, 'Metatem: A framework for programming in temporal logic'. In: *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*.

4. Beer, I., S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh.: 2001, 'The temporal logic Sugar'. In: *Proceedings of CAV 2001*.
5. Börger, E. and G. D. Castillo: 1995, 'A formal method for provably correct composition of a real-life processor out of basic components'. In: *Proceedings of the 1st ICECCS'95*. pp. 145–148.
6. Bowen, J., H. Jifeng, and X. Qiwen: 2000, 'An Animatable Operational Semantics of the VERILOG Hardware Description Language'. In: *Proc. of ICFEM2000: 3rd Int. Conf. on Formal Engineering Methods*. pp. 199–207.
7. Cau, A., C. Czarnecki, and H. Zedan: 1998, 'Designing a Provably Correct Robot Control System using a 'Lean' Formal Method'. In: A. P. Ravn and H. Rischel (eds.): *Proceedings 5th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT'98)*, Vol. 1486 of *LNCS*. pp. 123–132.
8. Cau, A. and H. Zedan: 1997, 'Refining Interval Temporal Logic specifications'. In: M. Bertran and T. Rus (eds.): *Transformation-Based Reactive Systems Development*, Vol. 1231 of *LNCS*. pp. 79–94.
9. Celoxica Ltd., 'Handel-C Language Reference Manual'.
10. Chen, W.-H., C. H. Smith, and S. C. Fralick: 1977, 'A fast computational algorithm for the discrete cosine transform'. *IEEE Transaction on Communications* **25**(9), 1004–1009.
11. Clarke, E. M., E. Emerson, and A. Sistla: 1986, 'Automatic Verification of finite-state concurrent systems using temporal logic specifications'. *ACM TOPLAS* **8**(2).
12. D.Borrione, P. Camurati, J. Piallet, and P. Prinetto: 1988, 'A functional approach to formal hardware verification'. In: *Proc. of ICCD-88*. pp. 592–595.
13. de Roever, W.-P., F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers: 2001, *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press.
14. Dimitrov, J.: 2000, 'Interval Temporal Logic (ITL) Semantics for Verilog'. Proc. of IEE event on Hardware-Software Co-Design.
15. Dimitrov, J.: 2001, 'Operational Semantics for Verilog'. In proceedings of APSEC'2001.
16. Dreike, P. and J. McCoy: 1997, 'Cosimulating Hardware and Software in Embedded Systems'. In: *Proceedings Embedded Systems Programming Europe*. pp. 12–27.
17. Ernst, R.: 1998, 'Codesign of Embedded Systems: Status and Trends'. *IEEE Design & Test of Computers* pp. 45–54.
18. Gajski, D. D., J. Zhu, and R. Domer: 1997, 'Essential Issues in Codesign'. Technical Report ICS-TR-97-26, University of California, Irvine, Department of Information and Computer Science.
19. Gordon, M.: 1995, 'The Semantic Challenge of Verilog HDL'. In: *Proc. 10th Annual IEEE Symposium on Logic in Computer Science*. pp. 136–145.
20. Gordon, M. J. C. and T. F. Melham: 1993, *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press.
21. Grundy, J.: 1991, 'Window Inference in the HOL System'. In: P. J. Windley, M. Archer, K. N. Levitt, and J. J. Joyce (eds.): *The Proceedings of the International Tutorial and Workshop on the HOL Theorem Proving System and its Applications*.
22. Gupta, R. K. (ed.): 1997, *Special Issue on Hardware/Software Partitioning of Design Automation for Embedded Systems Journal*, Vol. 2. Kluwer Academic Publishers.

23. Hale, R.: 2001, 'Using ITL for Co-Design'. In: *Proc. of the Verification Workshop VERIFY'01*.
24. Hale, R. W. S.: 1988, 'Programming in Temporal Logic'. Ph.D. thesis, Computer Laboratory, Cambridge University, Cambridge, England. Appeared as technical report 173 in year 1989.
25. Hale, R. W. S. and H. Jifeng: 1994, 'A real-time programming language'. In: J. Bowen (ed.): *Towards verified systems*. Elsevie, pp. 115–130.
26. Hollander, Y., M. Morley, and A. Noy: 2001, 'The *e* language: A fresh separation of concerns.'. In: *Proceedings of TOOLS Europe 2001*.
27. Holzmann, G.: 1990, *Design and Validation of Computer Protocols*. Prentice-Hall.
28. Huibiao, Z., J. Bowen, and H. Jifeng: 2001, 'From Operational Semantics to Denotational Semantics for Verilog'. In: *Proc. of CHARME2001: the 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*.
29. Huibiao, Z. and H. Jifeng: 2000, 'A Semantics of Verilog using Duration Calculus'. In: *Proc. Intl. Conf. on Software: Theory and Practice*. pp. 421–432.
30. Hunt, W.: 1986, 'FM8501: A verified microprocessor'. In: *Proc of IFIP WG 10.2 Workshop: From HDL To Guaranteed Correct Circuit Designs*. pp. 85–114.
31. IEEE: 1995, 'IEEE Standard Hardware Description Language based on the Verilog(©) Hardware Description Language'. IEEE Standard 1364-1995.
32. ITL homepage, 'http://www.cse.dmu.ac.uk/˜cau/itlhomepage/'.
33. Jifeng, H. and Z. Huibiao: 2000, 'Formalising Verilog'. In: *Proc. IEEE Intl. Conf. on Electronics, Circuits and Systems*. pp. 412–415.
34. Kurshan, R. P.: 1994, *Computer-aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press.
35. Lavenier, D., P. Quinton, and S. Rajopadhye: 1999, 'Advanced Systolic Design'. In: *Chapter 5, Digital Signal Processing for MultiMedia Systems*. Eds. Parhi and Hishitani.
36. Madsen, J., J. Grode, P. Knudsen, M. Petersen, and A. Haxthausen: 1997, 'LYCOS: the Lyngby Co-Synthesis System'. *Design Automation for Embedded Systems* **2**(2), 195–235.
37. Manjunathaiah, M. and G. Megson: 1999, 'Design of Multi-phase Regular Arrays'. Technical Report RUCS/1999/TR/016/A, Computer Science, The University of Reading, U.K.
38. Manjunathaiah, M., G. Megson, S. Rajopadhaye, and T. Risset: 2001, 'Uniformization of Affine Dependance Programs for Parallel Embedded System Design'. In: *Proceedings of 2001 International Conference on Parallel Processing (30th ICPP'01)*. U. Politec. de Valencia, Spain.
39. Megson, G.: 1992, *An Introduction to systolic algorithm design*. Oxford University Press.
40. Moszkowski, B.: 1986, *Executing Temporal Logic Programs*. Cambridge, England: Cambridge University Press.
41. Moszkowski, B.: 1994, 'Some very compositional temporal properties'. In: E.-R. Olderog (ed.): *Programming Concepts, Methods and Calculi*, Vol. A-56 of *IFIP Transactions*. pp. 307–326.
42. Nielson, F., H. Nielson, and C. Hankin: 1999, *Principles of Program Analysis*. Springer-Verlag.
43. Page, I.: 1996, 'Constructing hardware-software systems from a single description'. *Journal of VLSI signal processing* **12**, 87–107.
44. Sagdeo, V.: 1998, *The Complete Verilog Book*. Kluwer Academic Publishers.

45. Soininen, J.-P., T. Huttunen, K. Tiensyrja, and H. Heusala: 1996, 'Cosimulation of real-time control systems'. In: *Proceedings of the European Design Automation Conference with EURO-VHDL '95*.
46. Spivey, J.: 1997, 'Deriving a Hardware Compiler from operational semantics'. Technical report, Oxford University Computing Laboratory.
47. Thomas, D. and P. Moorby: 1998, *The Verilog Hardware Description Language*. Kluwer Academic Publishers.
48. Zhou, S., H. Zedan, and A. Cau.: 1999, 'A Framework For Analysing The Effect of 'Change' In Legacy Code'. In: *IEEE Proc. of ICSM'99*.

Address for Offprints:
Software Technology Research Laboratory
SERCentre, De Montfort University,
The Gateway, Leicester LE1 9BH, England
Tel.: +44(0) 116 257 7937
Fax.: +44(0) 116 257 7936