International Journal of Parallel Programming, Vol. 34, No. 1, February 2006 (© 2006)
DOI: 10.1007/s10766-005-0002-x

Verification Approach of Metropolis Design Framework for Embedded Systems

Xi Chen,[1,2,4] Harry Hsieh,[2] and Felice Balarin[3]

In this paper, we focus on the verification approach of Metropolis, an integrated design framework for heterogeneous embedded systems. The verification approach is based on the formal properties specified in Linear Temporal Logic (LTL) or Logic of Constraints (LOC). Designs may be refined due to synthesis or be abstracted for verification. An automatic abstraction propagation algorithm is used to simplify the design for specific properties. A user-defined starting point may also be used with automatic propagation. Two main verification techniques are implemented in Metropolis: the formal verification utilizing the model checker Spin and the simulation trace checking with automatic generated checkers. Translation algorithms from specification models to verification models, as well as algorithms of generated checkers are discussed. We use several case studies to demonstrate our approach for verification of system level designs at multiple levels of abstraction.

KEY WORDS: LTL; LOC; metropolis; meta-model; spin; property; simulation; formal verification.

1. INTRODUCTION

As Moore's Law continues its march through the millennium, design complexity of modern systems increases exponentially. Design and verification methodologies at higher levels of abstraction are required to fill the gap

[1]University of California, Riverside, California, USA. E-mail: {xichen, harry}@cs.ucr.edu
[2]Present address: Novas Software, Inc., San Jose, CA, USA.
[3]Cadence Berkeley Laboratories, Berkeley, CA, USA. E-mail: felice@cadence.com
[4]To whom correspondence should be addressed.

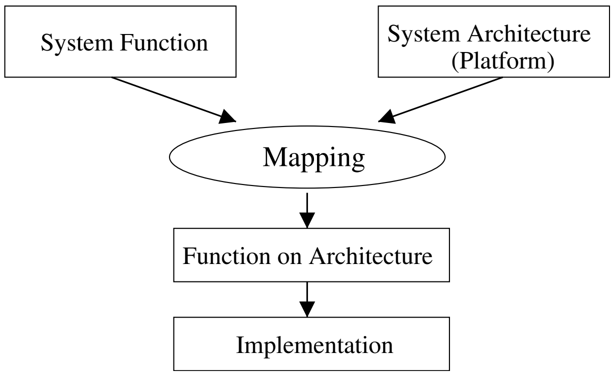

Fig. 1. System level design methodology.

between the increasing semiconductor manufacturing capabilities and the lag-behind design productivity. System level design, based on orthogonalization of design concerns, as well as pre-defined platforms, will become ever more important.(1) The initial specification of the function and the architecture of a system is done at a high abstraction level without particular lower level implementation details. The function is then mapped onto the architecture after iterations of refinement procedures (see Fig. 1). Significant advantages in design flexibility, as compared to the traditional fixed architecture and *a priori* partitioning approach, can result in significant advantages in the performance and design cost of the product.

Synthesis (i.e. steps taken toward implementation) is applied systematically to transform high level specifications into manufactured products. System level synthesis steps may include structural transformations, where designs are partitioned, composed, or otherwise altered, and formal refinements, where possible behaviors of the design, equivalently represented as non-deterministic traces, are formally refined through the use of constraints or implementation annotations. There exist multiple levels of abstraction in a design flow, which also indicate the requirement for suitable verification techniques to be applied at each level. Furthermore, abstraction is a basic operation to manage complexity during verification procedures. The tendency is to simplify (or abstract) the design for verification purposes and refine the design as more implementation details are determined.

The Metropolis framework is developed as an integrated and unified design environment for embedded systems.(2) Metropolis allows designers to represent and manipulate their designs at multiple levels of abstraction and with multiple models of computation (MoC). Integrated into the Metropolis design framework is a set of back-end tools, with which one can simulate, synthesize, and verify a design at hand. Central to the framework is the Metropolis Meta-Model (MMM) language. Different high level languages, MoC, design constraints, as well as specifications of system functions, architecture platforms and function-architecture mappings, can be represented in MMM while retaining their precise semantics. Constructs in MMM are chosen to facilitate the transformations and refinements between different abstraction levels. This paper focuses on the verification techniques integrated in Metropolis for system level designs. They are based on the formal specification of design properties and constraints in MMM. Designers are able to specify both functional and performance properties of a design in mathematical logics, Linear Temporal Logic (LTL)(3) and Logic of Constraints (LOC),(4) with MMM language constructs, and use the integrated verification tools to verify these properties in the design flow.

In property-based verification, only a portion of the design may be relevant to the passing or failing of a given property. The rest of the design may be simplified or removed, without changing the outcome of the verification. Based on this observation, a technique of automatic design abstraction and propagation is developed to abstract the original specification of a design and to simplify the corresponding verification model. Designers are also allowed to indicate what elements in the design are not relevant to the properties being verified. They can apply these abstraction operations, freeing in particular, to the variables, statements, and components. If the properties are "safety" in nature (i.e. something bad will never happen), the abstraction can only lead to the verification result that is either exact or conservative (with possibly false negative result). There will never be a false positive result. We propose an automatic algorithm to propagate this abstraction to the rest of the design exactly (i.e. without introducing more false negatives or any false positives).

For small but important designs or library modules that will be instantiated many times across different designs, it is possible and useful to exhaustively prove the desired properties at a high level of abstraction using formal verification techniques. In Metropolis, the model checker Spin(5) is utilized as one of its back-end verification engines. A design specification in MMM is automatically translated into a verification model in Promela, the modeling language of Spin, and the properties of the design can then be formally verified with Spin. We illustrate the usefulness of the formal verification methodology with a case study of a task transition level (TTL) FIFO channel. We then apply the technique of design abstraction and abstraction propagation to the same design example to demonstrate that such a "pre-processing" at the system level indeed leads to significantly shorter verification time.

Formally verifying practical implementations, not just their abstractions, is computationally difficult due to the "state explosion" problem. Simulation remains a primary means of verification when designs are refined with more implementation details. In Metropolis, a simulation verification methodology based on trace analysis of formal properties, is integrated into the back-end SystemC simulator. From the formal properties specified by designers, trace analysis tools or simulation monitors can be automatically generated to check the properties off-line or concurrently during the simulation. A complex design where the function is mapped to an architecture is used to show the usefulness of the simulation verification methodology.

The rest of this paper is organized as follows. In the next section, we review the Metropolis design framework, its design methodology and the MMM language. In Section 3, we introduce the technique of automatic design abstraction and propagation that can be used to simplify the verification problem. In Section 4, we present the formal verification methodology in Metropolis utilizing the model checker Spin. The usefulness of the formal verification methodology and effectiveness of the automatic design abstraction technique are demonstrated through a real Metropolis design in Section 5. In Section 6, we discuss the simulation verification methodology incorporated in Metropolis. Section 7 summarizes the verification approach of the Metropolis design framework and concludes the paper.

2. METROPOLIS

In this section, we introduce the Metropolis design framework, its design methodology, and the syntactic and semantic features of MMM, the design specification language of Metropolis.

2.1. The Metropolis Framework and Design Methodology

The integrated design environment consists of a meta-model description language, MMM, a front-end that constructs internal representations and analyzes static network structures, and a set of back-end tools that are responsible for simulation, synthesis, verification, and other tasks. Constructs in MMM are designed to facilitate the transformations and refinements between different abstraction levels. Different high-level languages, MoC, design constraints, as well as specifications of system functions, architecture platforms and function-architecture mappings can be represented in MMM. Different design aspects are orthogonalized, such as computation versus communication, function versus architecture, and specification versus implementation. The design complexity can therefore

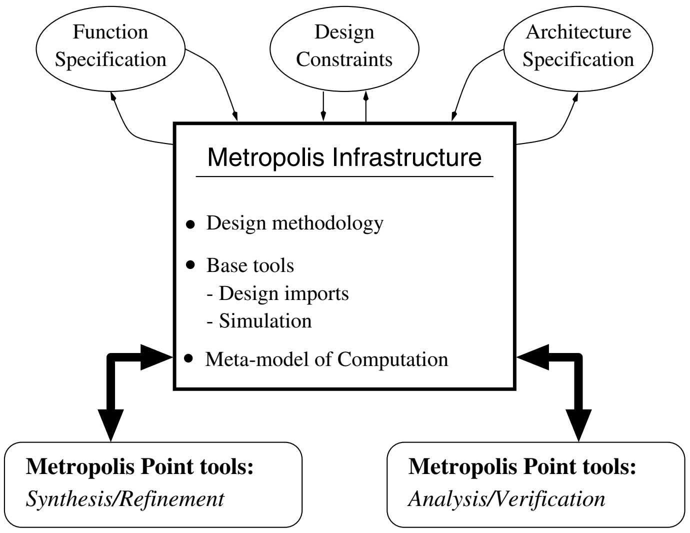
Function Specification
Design Constraints
Architecture Specification
Metropolis Infrastructure
Design methodology
Base tools
- Design imports
- Simulation
Meta-model of Computation
Metropolis Point tools: Synthesis/Refinement
Metropolis Point tools: Analysis/Verification

Fig. 2. Metropolis design framework.

be effectively reduced, and the design space can be efficiently explored. Figure 2 shows a flow diagram for the Metropolis framework.

2.2. The MMM Language

MMM is a system specification formalism capable of representing designs at different levels of abstraction. A description of a system (function and/or architecture) can be made in terms of computation, communication, and coordination.

2.2.1. Processes, Media, and Netlists

In MMM, systems are represented as networks of *processes* that communicate through *media*.(6) Processes and media are used to describe computation and communication respectively. The syntax of MMM is similar to Java but includes many system level extensions. A process defines an active object, and always includes a function called *thread* as the top-level function, where its behavior is specified. A communication medium implements a set of functions that are declared in *interfaces*. Processes connect to media through *ports*. Each port has a type, which must be an interface implemented by the medium to which the port is connected. Processes

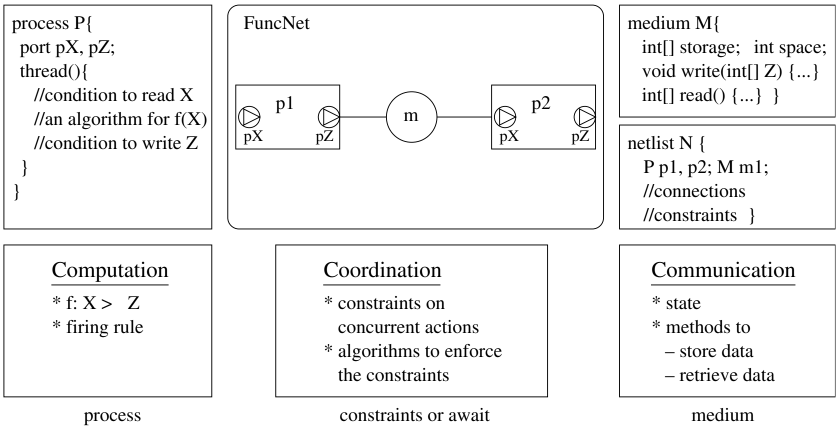

Fig. 3. An example of MMM specification.

communicate to each other by invoking interface functions implemented in the shared media through these ports.

In MMM, objects such as processes, media, and their connectivities can be grouped in a *netlist*, which is used to model a complete network. Figure 3 shows an example of a functional netlist *FuncNet*. The netlist defines two processes, *p1* and *p2*, communicating through a medium *m1*. A netlist can also contains other netlists to form a hierarchical network. In addition, refinement constructs are available to specify that one netlist is the formal refinement of another within a network.

2.2.2. Coordination

Processes run concurrently, each at its own pace. The relative speed of processes may arbitrarily change at any time, unless they synchronize with each other using the synchronization primitive called *await*, or if *constraints* are specified in the system. The await statement can be used to make a process wait until some condition holds and to establish critical sections that guarantee mutual exclusion among different processes. To limit the behavior of processes, a designer can also specify high-level LTL(3) or LOC(4) constraints, and leave the implementation of these constraints to the detail design stage.

The await statement is used to establish mutually exclusive sections and synchronize processes. It contains one or more statements called *critical sections,* each controlled by a triple (*guard*; *testlist*; *setlist*), where the guard is a Boolean expression, and the testlist and setlist denote sets of interface functions of other processes. A critical section is said to be *enabled* if its guard is true, and none of the interface functions in the testlist are being executed at that moment. A critical section may start executing only if it is enabled. In addition, while the critical section is being executed, no interface functions included in the setlist can begin their executions. Whenever an await is encountered in the execution flow, one and only one of the enabled critical sections is executed. If no critical section is enabled, the execution blocks. If more than one critical sections are enabled, the choice is non-deterministic.

2.2.3. Function, Architecture, and Mapping

The function-architecture separation and mapping is natively supported in the MMM language. System function and architecture are defined independently at a high level of abstraction. The function is then mapped to the architecture in order to arrive at a given implementation.

Both the function and the architecture of a system are modeled as separate networks of processes communicating through media. In an architectural network, resources are typically modeled with media, services that the architecture can provide are modeled with so called *mapping processes*, and arbitrators among multiple architectural resources are modeled with *quantities*. A third network can be defined to encapsulate the functional and architectural networks, and to *relate* the two by synchronizing events between them with *synch* constraints.

Figure 4 shows a mapping network *MapNetlist* that combines the functional network *FuncNetlist* with the architectural network *ArchNetlist*. The functional network includes two processes *p1* and *p2* communicating through media *m1* and *env*. The architectural network contains media *CPU*, *BUS* and *MEM*, and the corresponding mapping processes. The synch constraints are used to synchronize the events of functional processes and the mapping processes. Schedulers *OsSched* and *BusArbiter*, which are modeled with quantities, coordinate the architectural resources and provide performance models to the architectural network. During execution, architectural media and mapping processes can request the quantitative annotations from the quantities.

2.2.4. Functional and Performance Properties

Functional and performance design properties can be specified with the MMM language constructs in the form of LTL or LOC. LTL and LOC have different domains of expressiveness and indeed complement each other quite well.[7] At the verification stage, both static and runtime

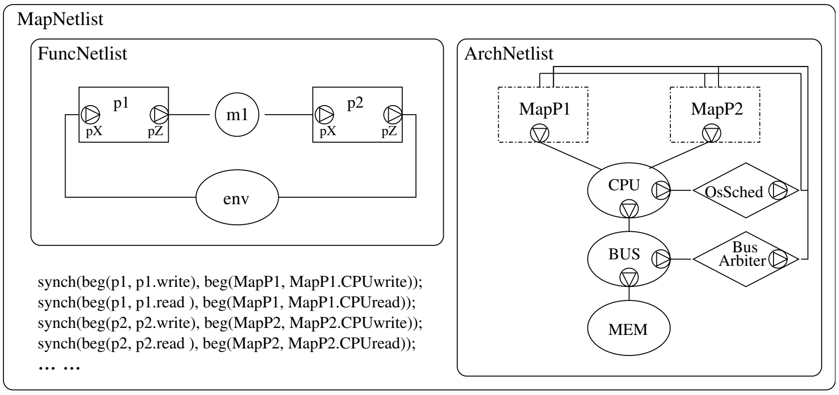
MapNetlist
FuncNetlist
p1
pX
pZ
m1
p2
pX
pZ
env
synch(beg(p1, p1.write), beg(MapP1, MapP1.CPUwrite));
synch(beg(p1, p1.read), beg(MapP1, MapP1.CPUread));
synch(beg(p2, p2.write), beg(MapP2, MapP2.CPUwrite));
synch(beg(p2, p2.read), beg(MapP2, MapP2.CPUread));
... ...
ArchNetlist
MapP1
MapP2
CPU
OsSched
BUS
Bus Arbiter
MEM

Fig. 4. Function-architecture mapping.

verification techniques can be used to check the design properties and report design errors if there is any property violation.

LTL is suitable for specification of functional properties, such as mutual exclusion, liveness and safety, and can effectively describe the temporal patterns for system state transitions. LTL formulas are constructed using terms, classical Boolean operators such as $\neg$ (not), $\vee$ (or), $\wedge$ (and) and $\rightarrow$ (imply), and the temporal operators G (globally), E (eventually) and U (strong until). Terms are Boolean conditions on variables or process states.

LOC is a formalism designed for specification of quantitative performance properties such as rate, throughput and latency, and functional properties such as I/O data consistency at the transaction level, where system events and their annotations are considered. It is also very well-suited for analyzing traces from execution of higher, transaction level system models. LOC consists of all the terms and operators allowed in sentential logic, with additions that make it possible to specify quantitative properties without compromising the ease of analysis. The basic components of an LOC formula include event names (e.g. *pipeline* and *sram_enq*), instances of events (e.g. $pipeline^{(4)}$), indices of event instances (e.g. 0, 1, ..., etc), the index variable i, and annotations (e.g. *cycle*, *pc* and *addr*). LOC can be used to specify many important system level performance properties that are inconvenient, and sometimes impossible, to specify with LTL. For example, the rate property:

$$cycle(pipeline[i+1]) - cycle(pipeline[i]) = 10 \tag{1}$$

requires that the difference between the values of annotation *cycle* for any two consecutive instances of *pipeline* event should equal to 10.

3. AUTOMATIC ABSTRACTION AND PROPAGATION

Working from a higher abstraction level of a design, such as Metropolis Meta-Model, provides two pivotal advantages. First, an abstraction applied to the higher level specification will also make the lower level verification model more abstract as well. It is therefore advantageous to apply abstraction operations directly on the higher level model and simplify the higher level model as much as possible. Second, designers now have the opportunities to specify abstraction operations on their own directly at the specification model according to their knowledge about the design.

It is obvious that only a portion of the design may be relevant to the passing or failing of a given property in property-based verification. The rest of the design may be simplified or removed, without changing the outcome of the verification. Unfortunately, identifying precisely what simplification or removal can be made correctly is as complex as the verification problem itself. Up until now, the process of design abstraction (i.e. the simplification of the design) is usually done by hand or left to the verification tools as they explore the reachable states and analyze the properties. Based on these observations, we propose an technique of automatic design abstraction and propagation to simplify specification models and to lead to a simpler verification models.

The automatic abstraction propagation consists of two separate operations, designer-driven propagation and property-driven propagation. Designers can specify free-able variables or statements according to their in-mind knowledge about the design, and then use the automatic propagation to exactly propagate them and abstract the entire design. The properties being formally verified may themselves suggest an exact abstraction as well. The property-driven propagation can automatically free the variables and statements that are not relevant. No designer's interaction is required.

The procedure of automatic abstraction propagation is illustrated in Fig. 5. If a regular verification session cannot complete or takes too much time to complete, a designer can turn on a compile-time flag to enable the abstraction, which can recognize the abstraction keywords and perform the automatic abstraction propagation to simplify the system design for verification. The abstraction propagation starts from the abstract syntax trees (ASTs), the internal representation of the Meta-Model language, uses on-demand traversal method to traverse the ASTs, and identifies the

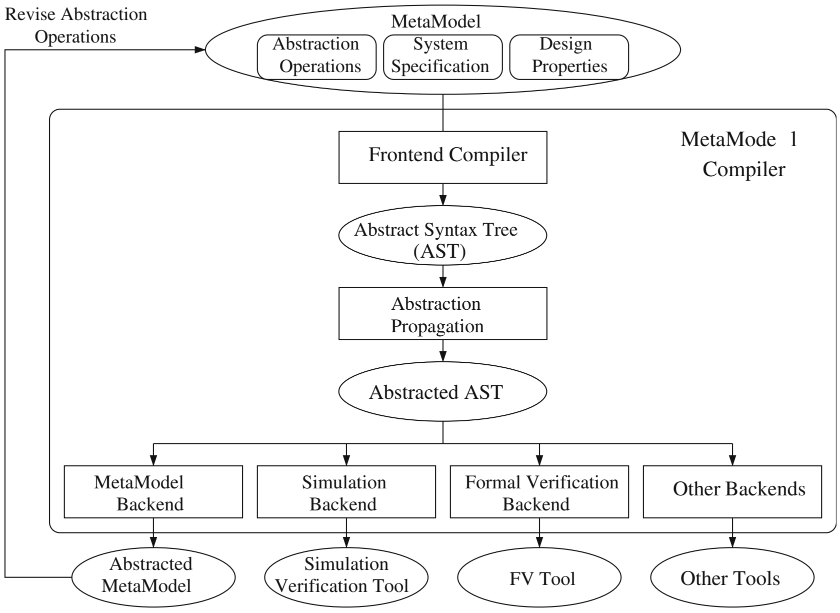

Fig. 5. Metropolis compiler architecture with abstraction propagation.

variables and statements that are eligible for abstraction according to the control and data dependencies in the design. Designers are allowed to specify more abstractions and the tool will propagate them automatically to abstract the design as much as possible to speed up verification. The abstracted specification can then be verified by other verification tools more efficiently.

3.1. Control and Data Dependency Graph

We use a control and data dependency graph (CDDG), built statically from the language syntax information of the original design, to automate abstraction propagation processes. The CDDG we use is a directed graph that has three types of vertices, representing variables, expressions and control statements respectively. More specifically, given an MMM specification, a CDDG is built according to the following general rules:

(1) Each variable in the design corresponds to a vertex in the graph. For each assignment expression, there is a expression vertex to represent its right-hand side expression. For each variable in the right-side expression, there is an edge from the variable vertex to the expression vertex. There is also an edge from the expression to the left-side variable.

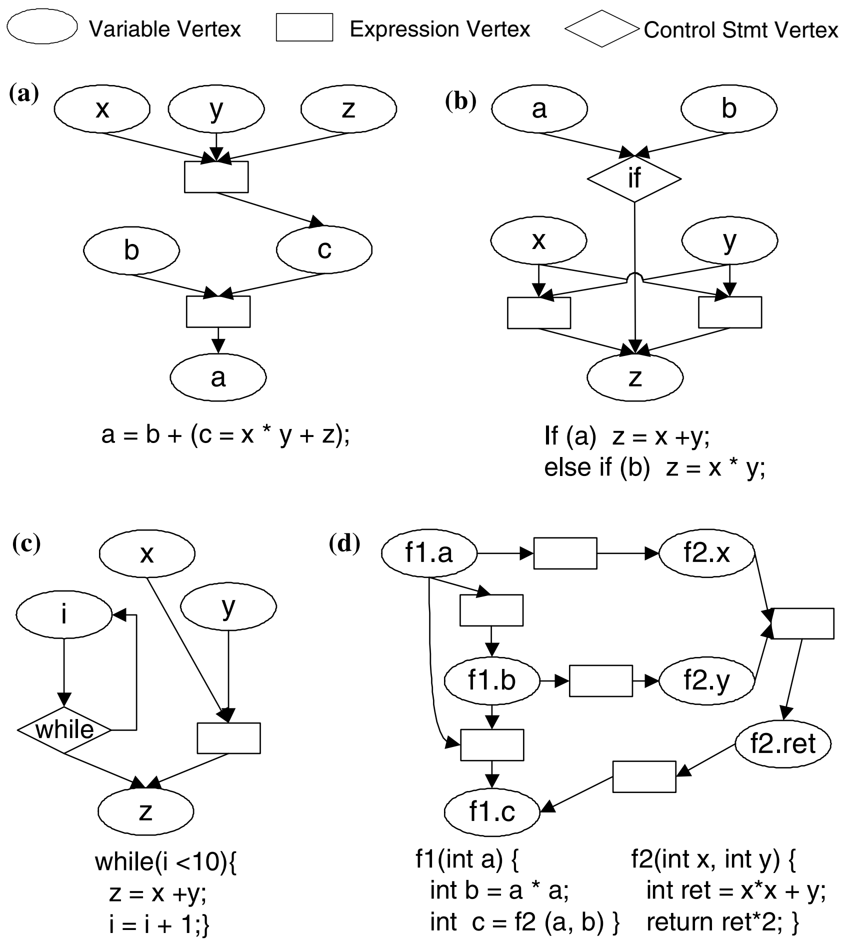

Fig. 6. CDDG examples.

Figure 6a shows an example of a complex assignment statement. Note that the operations on the variables are skipped and only dependencies are captured in a CDDG.

(2) Each control statement is represented as a vertex in the graph. If a variable is in the decision part of a control statement, there is an edge from the variable to the control statement; if a variable may change its value in the execution part of a control statement, there is an edge from the control statement to the variable. Control statements are further divided into three categories, branching statements such as *if* and *switch*,

loop statements such as *while* and *for*, and synchronization statements such as *await* and *synch*. Figure 6b and c shows examples of *if* and *while* statements respectively.

(3) Function calls are generally treated as operations and expressions. The CDDG of a design specification is built as if all of its functions are flattened. Functions are connected together through passing arguments and returning values when they invoke each other. For a function, there is an expression vertex for each of its formal parameters and an expression vertex for the return value. There are edges between the variable vertices (in both invoking and invoked functions) and these expression vertices as variables are passed as arguments and return values are assigned to variables. As Fig. 6d shows, function f_1 calls function f_2 by passing a and b as arguments and assigning the return value to c. So there are three expression vertices connecting the variables in two functions.

Note that vertices representing expressions don't contain any useful syntax information themselves and are only used to connect multiple variables to a variable or an expression as intermediate vertices. Though they are eventually removed to simplify the graph representation and traversal in the implementation, for the convenience of presentation, we still keep them in the illustrations. We define that a vertex v_j *depends* on a vertex v_i if there exists a directed path from v_i to v_j in a CDDG, where v_i and v_j represent either variables or control statements. In Fig. 6a, variable a depends on variables b, c, x, y and z. In Fig. 6c, variable i and the while loop depend on each other.

The number of vertices in a CDDG is the total number of variables, assignment expressions, formal parameters of functions and control statements. So the size of a CDDG is linear to the size of the original source code.

3.2. Abstraction Propagation Algorithms

Let $G = \{V,\ E\}$ be a CDDG that is built from a design specification, where V is the set of all the vertices and E is a set of dependency edges. In the designer-driven abstraction propagation, a designer can specify free-able variables and statements including variables and control statements, and automatically propagate them to the entire design. Assuming a set of variables and statements $D \subseteq V$ is chosen by the designer to start from for the designer-driven abstraction propagation, the algorithm is listed in Algorithm 1.

The algorithm searches for and then abstracts the variables and statements that depend on the designer's input in the entire specification. Using the example shown in Fig. 6c, if a designer specifies that the while loop

Algorithm 1 Designer-driven abstraction propagation.

```
1: $D' := D$
2: for each $v \in D$ do
3:     $D' := D' \cup \{u_i \in V$: there exists a path from $v$ to $u_i\}$
4: end for
5: Remove all the variables (including their operations) and statements in $D'$
   from the design.
```

Algorithm 2 Property-driven abstraction propagation.

```
1: $L := P$
2: for each $v \in P$ do
3:     $L := L \cup \{$all the vertices that have a path to $v\}$
4: end for
5: for each synchronization statement $s \in V$ do
6:     $L := L \cup \{s\}$
7:     $L := L \cup \{$all the vertices that have a path to $s\}$
8: end for
9: Remove all the variables (including their operations) and statements in $V - L$
   from the design.
```

is free-able, then the whole while loop including the variable i and the assignment statement of z will be totally abstracted and the abstraction can also be propagated to other variables and statements that depend on them. In the designer-driven abstraction propagation, the amount of false negative results due to the abstraction is decided solely by the designer's input. Its propagation is guaranteed to be exact and no false negative result will present as a consequence of the propagation.

Assume a set of variables $P \subseteq V$ is being checked in the properties. The algorithm of the *property-driven abstraction propagation* is listed in Algorithm 2. The algorithm keeps what the properties and synchronization statements depend on, and abstracts the rest of the design. Using the example shown in Fig. 6c, if $P = \{x, y\}$, $V - L = \{z\}$, the code fraction is then abstracted to: $\mathit{while}(i < 10)\ i = i + 1;$

Note that the synchronization statements are not freed at this point even if they don't directly control the variables in the properties. This is because a synchronization statement controls the execution of the processes in a concurrent system, and the complex interaction between processes make it difficult to free these synchronization statements exactly. The automatic abstraction propagation does not intend to handle the synchronization of concurrent systems and their abstractions are left to the designer by the designer-driven propagation. The algorithm also assumes that there are no non-terminating loops that may cause “dead” code.

Methodologically, the property-driven propagation should be applied first in the process of abstraction verification since it doesn’t need any interaction from the designer and will not introduce false negative results. Then a designer can apply several iterations of designer-driven abstractions to further abstract the design specification and simplify the verification problem as much as possible, even by introducing false negative results. The worst case for both algorithms is to traverse the entire CDDG $|V|$ times, so their complexity is $O(|V|^2)$ and they will introduce little overhead compared to the overall compilation time. The effectiveness of the automatic abstraction propagation we have proposed will be studied through a formal verification case study in Section 5.

4. FORMAL VERIFICATION FOR SYSTEM LEVEL DESIGNS

Metropolis allows designers to perform verification at different levels of abstraction, through an automatic translation mechanism that generates verification models from system specification models.

4.1. Formal Verification Methodology

The task of formal property verification is to exhaustively search the state space of a system design and to check whether a particular design property holds. After a system specification in MMM is translated to Promela description, one can use Spin to do property checking. Spin provides two powerful ways to specify properties of a design: Assertion and LTL properties.(3,8) Assertion is an annotation construct in Promela used to “assert” that a particular condition (e.g. space > 3) must hold. LTL is strictly a superset of Assertion properties. Without loss of generality, we only deal with the LTL formulas here.

The formal verification methodology of Metropolis is illustrated in Fig. 7. The MMM description is automatically translated into Promela description, and the LTL properties specified in MMM are checked using the model checker Spin. It is known that only a subset of LOC can be translated into equivalent LTL formulas and formally checked with Spin directly.(9) For the rest of the LOC formulas, the formal verification results may be inconclusive, i.e. the verification is only partial. The designer may perform any synthesis step (e.g. composition, decomposition, constraint addition, scheduler assignment) and a new Promela code can be automatically generated to verify the property. If it does not pass, the error trace may be used to help designers figure out whether the design needs to be

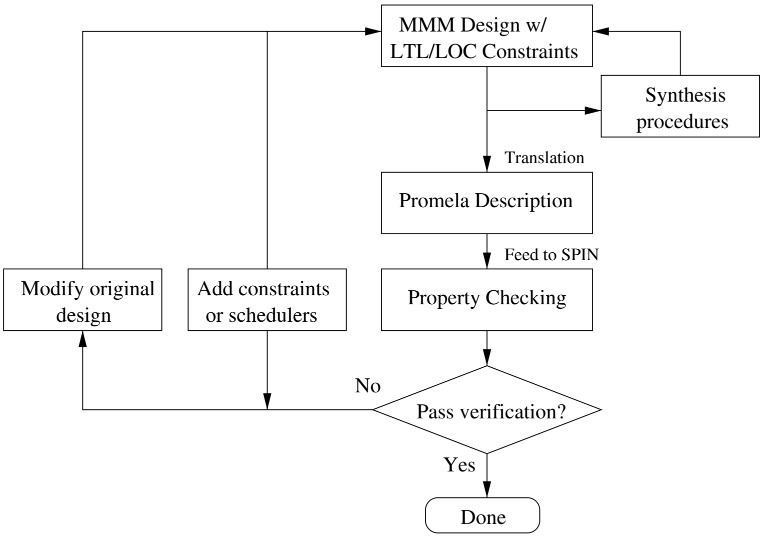

Fig. 7. Metropolis formal verification methodology.

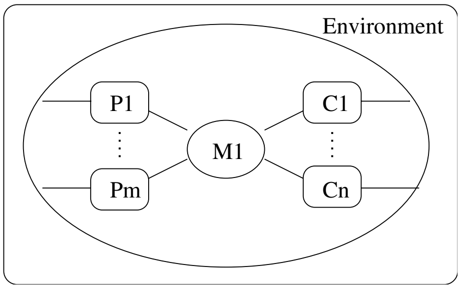

Fig. 8. Example of a bytelink meta-model.

altered. If the verification session runs too long, approximate verification can be used to explore a subset of the state space and report the probability that the property will pass. Obviously, a partial exploration cannot prove that a property holds. However, it is our experience that a lot of "easy" bugs can be found within a relatively small amount of time and memory usage. If a Spin verification session continues to run after a long time, it is highly likely that the property will eventually pass.

Figure 8 shows a prototypical network of m producers and n consumers communicating through a single medium. The producers receive inputs from the environment, process the data in some way, and then output it to a medium of a single space. The consumers read in information from that medium, process it, and then output to the environment. It is possible for all producers and consumers to execute concurrently. If we want to check the property, *"whenever a producer writes an item into the medium, there must be some space in the medium"*, it can be specified as an LTL formula:

$$\mathrm{G}((P_1\_write \vee \cdots \vee P_m\_write) \rightarrow M_1\_not\_full), \qquad (2)$$

and be verified with Spin after the specification model is automatically translated into Promela.

The same methodology can also be used for a verification-driven synthesis methodology. If the property does not pass the verification, an error trace is generated and examined. Based on the error trace, the original design may be incorrect, or refinement may be applied to the original specification for it to have the desired property. At a higher level of abstraction, constraints may be used to constrain the behavior so the property may pass. At a lower level of abstraction, designers must ensure that these constraints are implemented. This may be achieved, for example, with the schedulers on a particular platform.

4.2. Implementation

To formally verify an MMM design with Spin, the MMM specification needs to be translated to Promela description. The main constructs of MMM are processes, media, netlists, interfaces, await statements and *synch* constraints for function-architecture mapping. In MMM, computation is usually modeled as functions defined in processes, and communication between processes is made by calling functions defined in media. We use a translation approach that translates each MMM process to a Promela process, and in-lines all the functions into the process that calls them directly or indirectly. The translator simply pastes its translated code to the point of the invocation in the calling process. In the situation of multiple level function calls, all the functions are in-lined recursively so that one MMM process corresponds to only one Promela process. With function in-lining, the verification becomes much more efficient regarding both time and memory usage compared to the dynamic function invocation.

In MMM, an interface is used to define the I/O data ports of the process or medium and the I/O control points of the process or medium. To implement the control point, the MMM interface is used as a semaphore in the *setlist* and *testlist* of an *await* statement. We translate each interface into a pair of integer variables used as semaphores in Promela. The first variable, called *ACTIVE* is used to indicate whether the interface (and its member functions) are in active state (whether they are being executed). Another one called *EXCLUSIVE* indicates whether this interface semaphore is set (i.e. whether it is included in the *setlist* of some *await* statement that is currently executing). We use these variables as semaphores to signal that interface functions appearing in *testlist*'s are being executed and to prevent, when appropriate, interface functions appearing in *setlist*'s from being executed. Promela constructs such as *atomic*, repetition *do-od* and case selection *if-fi* are utilized to guarantee the exact semantics equivalence of *await* statements. Specially, if the *await* statement has more than one critical sections that are enabled, one of them will be chosen non-deterministically and executed. This non-determinism is directly supported in Promela in *do-od* and *if-fi* statements.

Another interesting aspect of MMM is the existence of dynamic objects (i.e. references). For example, an array is represented by a reference in MMM, and its memory space could be allocated and changed dynamically at runtime. However, most model checkers (including SPIN) only support static memory allocation, i.e. arrays have to be declared explicitly at design time. To solve the problem, we have to put some restrictions on the MMM code. All the reference types have to be declared explicitly once and only once, so that they can be translated to Promela as static objects. An array declaration in MMM, "$int[]\,a = new\,int[12];$" can be translated to Promela as a static array "$int\ a[12];$". After the array a is declared in MMM, its reference cannot be changed any more. If the dimension of the MMM array is dynamic, e.g. "$int[]\ a\ =\ new\ int\ [n];$" where n is a variable, it is also translated to Promela as a static array "$int\ a[ARRAY\_MAX];$", where $ARRAY\_MAX$ is a constant set by the designer at the compilation time. It is up to the designer to guarantee that $ARRAY\_MAX$ is always larger than or equal to the maximum value of n. Other dynamic objects such as class types in MMM are similarly translated to static data objects of Promela.

In MMM, the function of a system is specified as processes communicating through media. The architecture is represented as a set of media and mapping processes. Synchronization constraints are used to map the function to the architecture. To translate the function-architecture mapping, we need to use Promela to implement the MMM synchronization constraints that actually relate the function processes and the architectural mapping processes together. In Promela, we use a rendezvous channel (or synchronous channel) to synchronize two concurrent processes. Using an example of a MMM synchronization constraint from Fig. 11:

"$synch(beg(p1, p1.write), beg(MapP1, MapP1.CPUWrite));$ ", the beginning of *p1*'s *write* and the beginning of *MapP1*'s *CPUWrite* are synchronized (both *write* and *CPUWrite* are function calls). In Promela, when either *p1* or *MapP1* runs to a point that needs to be synchronized, it sends a synchronization signal to a rendezvous channel, and waits for the other process. In this way, the events of the functional processes and their corresponding mapping processes are synchronized, and mapping is realized in Promela.

5. A FORMAL VERIFICATION CASE STUDY

In this section, we use a realistic Metropolis design as an example to illustrate the usage of the formal verification mechanism in Metropolis and to demonstrate the effectiveness of the automatic abstraction propagation we have proposed in Section 3.

Y-chart Application Programmer's Interface (YAPI) is a popular model of computation for designing signal processing systems.(10) It is basically a Kahn process network(11) extended with the ability to non-deterministically select an input port to consume and an output port to produce. Within Metropolis, a library environment is set up such that any YAPI design can be written using constructs in the Metropolis library. Central to YAPI is the definition of communication channel and its refinement into TTL.(12,13) Figure 9 shows how a YAPI channel is refined to a TTL channel in Metropolis. A YAPI channel models an unbounded First-In-First-Out (FIFO) buffer, similar to Kahn process network. Asynchronously, writer processes write data into one end of the channel and reader

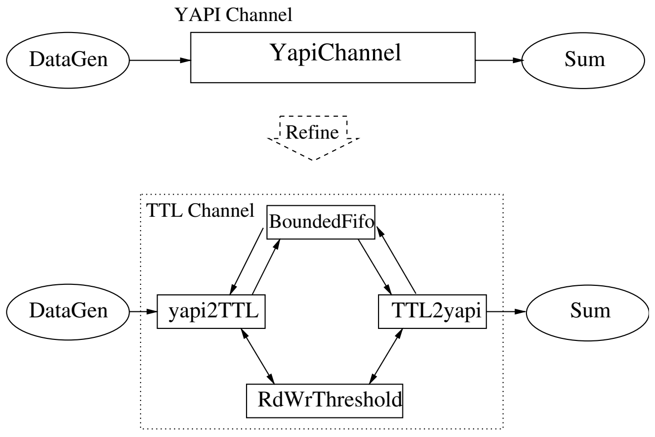

Fig. 9. YAPI and TTL channels.

processes read data from the other end of the channel. At the lower level (TTL), the channel is modeled with a bounded FIFO buffer. A central protocol is used to control the mutual exclusion and boundary checking of the bounded FIFO buffer. As Figure 9 shows, the TTL channel has a bounded FIFO (*BoundedFifo*) whose size is set at design time, and a control medium (*RdWrThreshold*) which implements a protocol to guarantee correctly writing to and reading from the FIFO buffer. To test the YAPI channel and its TTL refinement, we use a writer process (*DataGen*) to write a series of data into the channel and a reader process (*Sum*) to read the data from it.

Due to the boundedness of the TTL buffer, the writer process will block when there is not enough free buffer slots to write data, and the reader process will block when there is not enough data available in the buffer. The protocol implemented in the TTL channel controller(*RdWrThreshold*) uses a threshold value to indicate if the writer or the reader can be unblocked. If there is a condition on which a process may be unblocked, the controller uses events *wakeup_reader* or *wakeup_writer* to signal unblocking. The detail of this algorithm can be found in Ref. 12. The TTL channel model has 720 lines of code in MMM and about 2200 lines code in Promela after translation. The experiments presented in this paper are all conducted with Spin 4.1.3 on our 3.0 GHz Pentium 4 machine with 4 GB of total memory.

One important property we want to check on the TTL channel is that there should be no deadlock situation within the channel, i.e. once the writer starts writing data into the channel, it will finish writing eventually. This property can be specified as an LTL formula:

$$\mathtt{G}(datagen\_start \rightarrow (\mathtt{E}\ datagen\_finish)), \tag{3}$$

where G is the *globally* operator and E is the *eventually* operator in LTL.

First, we try to verify a preliminary version of the TTL channel that contains a real bug causing a deadlock situation. Using Spin, the bug can be easily caught within less than 1 min.[5] Then, after fixing the bug, we re-run the verification session and the revised TTL model can pass the formal verification without any error. The total CPU time used for the verification is a little less than 12 h. Table I lists the details about the verification sessions for the non-deadlock property of the TTL model with and without abstractions and propagations applied.

[5]After the abstractions and their propagations are applied later, the bug in the preliminary TTL channel can also be caught within less than 1 min. So the abstractions and their propagation are considered safe.

Table I. Summary of Formal Verification for TTL Channel

	Verification w/o abstraction	Manual abstraction	Designer-driven abstraction prop.	Property-driven abstraction prop.
State vector	432 bytes	352 bytes	232 bytes	188 bytes
Depth reached	75607	74073	54359	33897
States generated	2.36686e+09	2.36607e+09	2.26572e+09	2.26481e+09
State transitions	3.65231e+09	3.60348e+09	3.42441e+09	3.54922e+09
Memory usage	1094.545 MB	1091.66 MB	1086.046 MB	1081.028 MB
CPU time usage	11 h:48 m:51s	10 h:26 m:24 s	6 h:41 m:03 s	5 h:37 m:24 s
Hash factor	3.62926	3.63046	3.79126	3.79278

*Optimization techniques, partial order reduction and bitstate, are applied.

Considering that the non-deadlock property only checks the control part of the TTL channel, its data-path can be abstracted to reduce the verification complexity. So we first manually free the data storages in both the writer process (*DataGen*) and the reader process (*Sum*) without using the automatic abstraction propagation. This abstraction saves about 12% of verification time, and requires modifying more than 10 statements throughout the original design. Then we use the designer-driven abstraction propagation to propagate these two abstractions to rest of the design. As a result, the internal data-path in the TTL channel is also abstracted and 43% of the verification time is saved.

To show the effectiveness of the property-driven automatic abstraction propagation, we also apply it on the original design. It automatically frees not only the FIFO structure but also the buffers in other two connecting components (*yapi2TTL* and *TTL2yapi*), which are directly connected to the FIFO, and their operations. From Table I, we can see the property-driven abstraction propagation can save 52% of verification time without any human interaction.

Practically, the designer-driven and property-driven abstraction propagation complement each other and should be used together to simplify the verification problem as much as possible.

6. SIMULATION VERIFICATION IN METROPOLIS

A simulation verification methodology based on trace analysis for design properties is integrated in the Metropolis simulator.

6.1. Simulation Verification Methodology

Figure 10 illustrates the methodology of the simulation verification based on trace analysis and automatic trace checker generation. The methodology begins with the formal specification of LOC or LTL properties in MMM, automatically generates runtime monitors or static checkers for trace analysis, checks the simulation traces during or after the simulations, and reports design errors if there is any property violation. According to the error report, a designer can either correct the original design or revise the property specifications until the trace analysis passes the verification.

LTL is defined over *executions* of a system, i.e. linear sequences of *state transitions*. In the simulation based trace analysis for system level models, the state transitions are modeled as event occurrences. This is consistent with transaction abstraction since only the event orderings are considered, not their tick by tick, cycle level behavior. We transform LTL formulas specified in MMM into the standard property specification language Sugar2.0,[14] leverage the existing tool, FoCs,[15] to generate the checker core, and then use our tool to automatically generate wrappers that are necessary for the simulation monitors and for static trace checkers. Since the simulation sessions are finite, we interpret the temporal operators over the finite traces by checking the conditions only up to the end of the traces.

A stand-alone automatic tool has been developed to generate checker cores for trace checkers or simulation monitors for given LOC formulas. A checker or monitor evaluates the formula instance by instance, where a formula instance is a formula with i evaluated to some fixed positive integer value. For example, $cycle(pipeline[30]) - cycle(pipeline[29]) = 10$ is the 29th instance of the formula (1). The details of the checking algorithms and data structures for LOC can be found in Ref. 9.

6.2. A Case Study

We use a high level model of function-architecture mapping to demonstrate the usage of the simulation verification methodology based on trace analysis.

In the platform-based design, mapping is the key procedure that correlates the function to the architecture. In this design example (as shown in Fig. 11), two source processes (S1 and S2) write the data into two independent channels. A separate process (Join) then reads data items from both channels, manipulates them, and then sends the result data to another process (Sink) through another channel. In the abstract architecture model, there are two CPU/RTOS units, a bus unit, a memory unit and a quantity

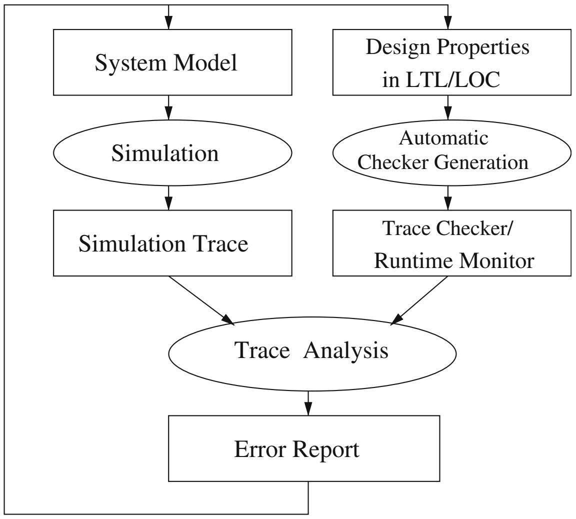
System Model
Design Properties in LTL/LOC
Simulation
Automatic Checker Generation
Simulation Trace
Trace Checker/ Runtime Monitor
Trace Analysis
Error Report

Fig. 10. Simulation verification methodology based on trace analysis.

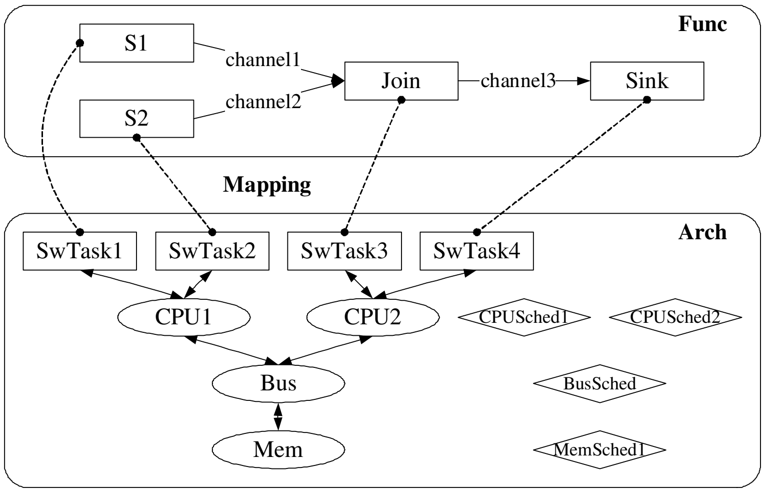

Fig. 11. A mapping model.

manager (i.e. scheduler) for each architectural unit.[6] A CPU unit can be shared among several software tasks that may request services from it. When more than one service request is issued to a CPU, arbitration is needed.

[6]An architectural unit is modeled as a medium in Metropolis.

The mapping procedure synchronizes the processes in the function model and the mapping processes (representing software tasks) in the architecture model. In this example, functional processes S1 and S2 are mapped to mapping processes SwTask1 and SwTask2, respectively, which are associated to CPU1 and the other two processes are mapped to CPU2. The CPU quantity managers implement a non-preemptive static-priority dynamic scheduling policy. The two CPU units are connected to the bus and the bus is connected to the memory unit. During simulation, the functional events are time-stamped through the architecture model, and thus various performance properties can be analyzed. With the sample input we used, the simulation took 14 min and produced a 1.1 G trace file with 2.36×10^7 lines.

We analyze the throughput of the model by using the LOC formula:

$$time(Sink\_read[i+100]) - time(Sink\_read[i]) \leqslant 5.0 \times 10^{-5}, \tag{4}$$

where event *Sink_read* represents the read operation by process Sink. The formula passes the trace verification in less than 1 min, which means process Sink can perform at least 100 read operations in every time period of 5.0 ns.

Similarly, we can check the latency between the source processes and process Sink by checking their events representing write and read operations respectively:

$$time(Sink\_read[i]) - time(S1\_write[i]) \leqslant 1.5 \times 10^{-7}, \tag{5}$$

and

$$time(Sink\_read[i]) - time(S2\_write[i]) \leqslant 1.5 \times 10^{-7}. \tag{6}$$

We can also analyze the processing delay of the Join process using the formula:

$$time(Jion\_write[i]) - time(Join\_read[i]) \leqslant 5.0 \times 10^{-7}. \tag{7}$$

It should be emphasized that timing is only one of the possible annotations we can use to analyze quantitative properties of a design. Any values associated with events can be used as annotations to check corresponding properties (e.g. data value or power).

In addition, LTL formulas can be used to verify temporal properties of the events generated by different processes (e.g. the event order). For example, the property that process Join cannot read before both source processes write and process Sink cannot read before process Join writes can be verified with the formula:

$$\begin{aligned} &\mathtt{G}\ ((\neg Join\_read\ \mathtt{U}\ (S1\_write\ \wedge\ S2\_write)) \\ &\quad \wedge(\neg Sink\_read\ \mathtt{U}\ Join\_write)). \end{aligned} \tag{8}$$

Given the trace size, all these property formulas can be analyzed within 1 min.

7. CONCLUSIONS

In this paper, we have presented the verification methodologies integrated in the Metropolis design framework for system level designs. Based on the formal specification of design properties in Metropolis, both formal verification and simulation verification can be applied to verify designs at multiple levels of abstraction. The technique of automatic design abstraction and propagation has been proposed to simplify property-based verification problems. Case studies have been performed to show the power of these verification techniques. Future work includes abstraction for simulation verification. The goal of abstraction for simulation is to significantly reduce simulation time for large designs.

ACKNOWLEDGMENTS

We acknowledge team members of the Metropolis project led by Prof. Sangiovanni-Vincentelli from University of California, Berkeley. The work presented here has been done within the framework of the Metropolis project, and it has benefited greatly from many discussions with team members.

REFERENCES

1. K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, System Level Design: Orthogonalization of Concerns and Platform-based Design, *IEEE Trans, Computer-Aided Design,* **19**(12):1523–1543 (December 2000).
2. F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, Metropolis: An Integrated Electronic System Design Environment, *IEEE Comput* **36**(4):45–52 (April 2003).
3. P. Godefroid and G. J. Holzmann, On the Verification of Temporal Properties, *Proceedings of IFIP/WG6.1 Symposium on Protocols Specification, Testing, and Verification* (June 1993).

4. F. Balarin, Y. Watanabe, J. Burch, L. Lavagno, R. Passerone, and A. Sangiovanni-Vincentelli, Constraints Specification at Higher Levels of Abstraction, *Proceedings of International Workshop on High Level Design Validation and Test* (November 2001).
5. G. J. Holzmann, The Model Checker Spin, *IEEE Trans. Software Eng.*, **23**(5):279–258 (May 1997).
6. F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, M. Sgroi, and Y. Watanabe, Modeling and Designing Heterogeneous Systems, *Technical Report 2001/01 Cadence Berkeley Laboratories* (November 2001).
7. X. Chen, H. Hsieh, F. Balarin, and Y. Watanabe, Verifying LOC Based Functional and Performance Constraints, *Proceedings of International Workshop on High Level Design Validation and Test* (November 2003).
8. Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer-Verlag (1992).
9. X. Chen, H. Hsieh, F. Balarin, and Y. Watanabe, Logic of Constraints: A Quantitative Performance and Functional Constraint Formalism, *IEEE Trans Computer-Aided Design Integrated Circuits*, **23**(8):1243–1255 (August 2004).
10. E. d. Kock, G. Essink, W. Smits, P. v. d. Wolf, J. Brunel, W. Kruijtzer, P. Lieverse, and K. Vissers, YAPI: Application Modeling for Signal Processing Systems, *Proceedings of the* 37^{th} *Design Automation Conference*, (June 2000).
11. G. Kahn, The Semantics of a Simple Language for Parallel Programming, *Proceedings of IFIP Congress*, North Holland Publishing Company pp. 471–475 (1974).
12. J. Brunel, E. A. de Kock, W. M. Kruijtzer, H. J. H. N. Kenter, and W. J. M. Smits, Communication Refinement In Video Systems on Chip, *Proceedings of the 7th International Workshop on Hardware/Software Codesign*, pp. 142–146 (1999).
13. O. Gangwal, A. Nieuwland, and P. Lippens, A Scalable and Flexible Data Synchronization Scheme for Embedded HW-SW Shared-Memory Systems, *Proceedings of International Symposium on System Synthesis* (October 2001).
14. C. Eisner and D. Fisman, Sugar 2.0 Proposal Presented to the Accellera Formal Verification Technical Committee (March 2002).
15. Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar, and Y.Wolfsthal, FoCs-Automatic Generation of Simulation Checkers from Formal Specifications, *Technical Report, IBM Haifa Research Laboratory, Israel* (2003).