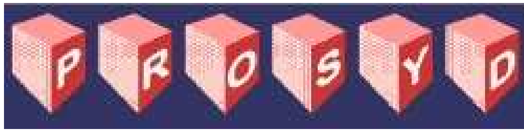

FP6-IST-507219

PROSYD:

Property-Based System Design

Instrument: Specific Targeted Research Project

Thematic Priority: Information Society Technologies

Property-Based Design and Implementation (Deliverable 2.1/1)

Due date of deliverable: May 24, 2005 (reissue)
Actual submission date: 19 May 2005

Start date of project: January 1, 2004 Duration: Three years

Organisation name of lead contractor for this deliverable: TUG

Revision 2.0

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	☒
PP	Restricted to other programme participants (including the Commission Services)	☐
RE	Restricted to a group specified by the consortium (including the Commission Services)	☐
CO	Confidential, only for members of the consortium (including the Commission Services)	☐

Notices

For information, contact Roderick Bloem rbloem@ist.tugraz.at.

This document is intended to fulfil the contractual obligations of the PROSYD project concerning deliverable 2.1/1 described in contract number 507219.

The information in this document is provided ”as is”, and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

© Copyright PROSYD 2005. All rights reserved.

Table of Revisions

Version	Date	Description and reason	By	Affected sections
0.1	March 2004	Initial draft. Summary about property-based synthesis	Moulin, Auerbach	All
0.2	June 2004	Changed structured and added technical approach for property-based synthesis	Jobstmann, Bloem	All
0.3	July 2004	Worked in comments form partners comments	Jobstmann, Bloem	All
1.0	August 2004	Submission to committee	Jobstmann	All
1.1	March 2005	Complete rewrite to conform to reviewer comments as stated in review report received Feb 22, 2005	Jobstmann, Bloem	All
1.2	April 2005	Major revision in the example sections.	Jobstmann, Bloem	2
1.3	May 2005	Reorganization of section 2, 3, and 4	Bloem	2,3,4
2.0	15 May 2005	Final approval by project coordinator	Eisner	-

Authors

Gadiel Auerbach
Mark Moulin
Barbara Jobstmann
Roderick Bloem

Executive Summary

This document describes the methodology of property-based error localization and property-based synthesis, which together form Workpackage 2.

Error localization addresses the problem of debugging. When a failure is detected during the verification of a block, finding the fault that causes it can be very time consuming. This lengthens the design cycle, often right before delivery time, when delays are especially bothersome. Error localization aims to automatically find and, if possible, correct the fault, thus freeing up designer resources for more complex and creative tasks such as design. We consider the design cycle of a block to consist of specification, design, and repeated iterations of detecting the presence of a fault followed by localization and correction. Automating the detection of a fault is the aim of verification tools; fault localization aims to automate the second step, and correction automates the last. The supervision of a designer remains necessary even for automatic correction, to make sure that the suggested correction adheres to unstated or non-functional requirements.

Property synthesis aims to shorten the design cycle by automating the process of designing a block. We identify the following benefits:

1. The need to hand-code blocks that are not area or timing critical is removed. Automatically generated blocks are guaranteed to fulfill their functional specifications.
2. Designers can develop their block in the presence of a fully operational environment because functional prototypes of blocks can be created right after specification. Thus, integration testing can start right away and a major source of design mistakes is circumvented. Also, prototypes can be used as a starting point for an efficient hand implementation.
3. Specification faults are found very early in the design process, as they are immediately apparent in the synthesized system. Thus, a correct specification is obtained earlier and incorrect implementations are avoided.

Both error localization and property synthesis aim to automate important parts of the design flow, shortening it and reducing time to market.

Purpose

This document is the final result of Task 2.1. It is intended to be an introduction to property-based design and implementation and should pave the way for the other tasks in Workpackage 2.

Intended Audience

This guide is intended for researchers working on verification tools using PSL or a similar specification language. It is assumed that readers are familiar with the notions and terms related to PSL and have a basic understanding of model checking terminology.

Background

Currently, there are no methods for fault localization based on temporal properties and work on sequential circuits is very rare as well. Most of the work focuses on a few particular faults such as a forgotten inverter. Existing technologies rely on a given reference model. We do not have this requirement because the property takes the place of the reference model.

The theoretical background of the synthesis problem is well-investigated but an implementation of a synthesis tool for LTL, PSL, or a similar logic does not currently exist.

Further background to error localization and property synthesis is given in Sections 2 and 3, respectively.

Contents

Table of Revisions iii
Authors iii
Executive Summary iii
Purpose iv
Intended Audience iv
Background iv
Contents v
Table of Figures vi
List of Tables vii
Glossary viii
1 Introduction 1
Motivation 1
Methodology Overview 2
2 Error Localization 5
Background 5
Methodology and Examples 6
Localization 6
Correction 9
Tool 11
Inputs and Outputs 11
Components 12
Technical Approach 12
Localization 12
Correction 13
3 Property Synthesis 15
Background 15
Methodology and Examples 15
Examples 15
Applicability 18
Tool 18
Inputs and Outputs 18
Components 19
Technical Approach 19
4 Conclusions 21
5 References 23

Table of Figures

Figure 1 - Time-saving by Fault Localization 2
Figure 2 - The PROSYD Tool Set 3
Figure 3 - Fault Localization 7
Figure 4 - A Simple Arbiter 7
Figure 5 - Manual Design 8
Figure 6 - Fault Correction 10
Figure 7 - 8-Bit Multiplier 11
Figure 8 - Synthesis 16
Figure 9 - Simple Design 16
Figure 10 - Synthesis of Blocks in Executable Environment 17
Figure 11 - Simple design for an incomplete property set 17

List of Tables

Table 1 - Property Set of an Arbiter with two Clients 8

Glossary

Abduction Based Diagnosis
In abduction based diagnosis it is known in which ways a component can fail. Using this knowledge, abduction based diagnosis tries to find a component of the model and a corresponding fault that explains the discrepancy between observed and desired behavior.

Atomic Proposition
Atomic proposition of a formula in a propositional logic. They correspond to the signals in a design.

Block
A group of interconnected cells. A block may contain instances of other blocks.

Combinational Circuit
A logic circuit whose output is a function of only the present input.

Consistency Based Diagnosis
Fault diagnosis based on the principle of solving contradictions.

Design
A netlist for a block, consisting of gates and the electrical connections between them (signals).

Error
The difference between a computed, observed or measured value or condition and the true, specified, or theoretically correct value or condition. For example, a difference of 30 meters between a computed result and the correct result.

Expansion Rules
The expansion rules split a PSL formula into two parts: a part that has to be true in the current time step and a part that has to be true in the next time step. For instance, the expansion rules state that $p \cup q = q \vee p \wedge \text{next}(p \cup q)$.

Failure
An incorrect result. For example, a computed result of 12 when the correct result is 10.

Fault
An incorrect step, process, or data definition. For example, an incorrect instruction in a computer program

Fault/Error Localization
The process of isolating parts of the design likely to be responsible for the failure of the design found in the verification step. "Fault localization" is the correct terminology but "error localization" is commonly used.

Fault/Error Correction
The process of correction a fault in a design. Again, "fault correction" is the correct terminology but "error correction" is commonly used.

Finite State System
A finite state system is a Mealy machine: it consists of latches and combinational logic with input and output, and abstracts from timing issues.

Formal Verification
Any mathematical method for verification, capable in principal of returning either "true" or "false" to a verification problem. In contrast, simulation or testing is an informal method of verification, as it can prove only the existence of a bug, but not the absence of one.

Gate
Another name for a logic cell, which is a functional group of transistors having physical attributes that support a specific semiconductor process technology.

Infinite Game
A finite state machine on which two players, the protagonist and the antagonist, determine an infinite run by each determining part of the input. The game comes with a winning condition and the task of the protagonist is to make sure that the run satisfies this condition.

Invariant
An invariant is a special case of a safety property. It sets a constraint on the state of the program that must hold in all states. For instance, "`a` and `b` are never 1 simultaneously."

Mistake
A human action that produces an incorrect result. For example, an incorrect action on the part of the programmer or operator.

Model Based Diagnosis
A formal method for fault localization. We distinguish between abduction based and consistency based diagnosis.

Integration Testing
We distinguish between unit tests, in which a single block is tested in isolation, and integration tests, in which blocks (usually from different designers) are tested in combination.

PSL
Property Specification Language, the language for specification of designs upon which PROSYD is based.

Safety Property
A safety property states that something bad should not happen. For instance, "a is never 1 in two consecutive clock ticks."

SAT Solver
An algorithm for solving instances of the Boolean satisfiability problem.

Sequential circuit
A finite state system.

Synthesis
The process of automatically generating a design from a given specification.

Unrolling
By unrolling a sequential circuit for a finite amount of time frames a equivalent combinational circuit is obtained.

Winning Strategy

A recipe with which a player is guaranteed to win an infinite game, no matter what the other player does. A finite state strategy may depend on a finite memory of the past, i.e., the move the strategy suggests can depend on previous moves of the two players. A memoryless strategy depends only on the current state of the game.

1 Introduction

In this document we address two aspects of property-based design, which are property-based error localization and property synthesis. Together they form PROSYD Workpackage 2. In the following, we motivate the need for property-based error localization and property-based synthesis and list their advantages. We conclude this section with an overview of how error localization and property-based synthesis are integrated in the design flow of electronic systems. In Section 2 we deal with property-based error localization. The section starts with background information. Then we describe the methodology of using error localization based on concrete examples. Finally, we talk about the error localization tool and explain our technical approach. Section 3 provides an insight into property-based synthesis. Again, we start with background and methodology followed by examples highlighting the benefits of property-based synthesis. We end with details of the property-based synthesis tool and the corresponding technical approach. In Section 4 we conclude.

Motivation

As design density and complexity of digital systems increase, the costs due to design faults increase exponentially. Therefore, detecting, localizing, and correcting faults are crucial issues in today's fast-paced and fault-prone development process.

Formal and dynamic verification tools detect faults and provide the user with a failing run. Even with a detailed failing run in hand, locating and correcting a fault is a bland and time-consuming chore. Debugging, which is the process of locating and correcting a fault, is not done solely by designers. The verification engineers, the ones who write and run the verification tests, usually spend quite a lot of their own time analyzing the failure traces themselves. This is done to filter out errors in the verification environment so that as few spurious traces as possible are passed on to the designers. Debugging is one of the most time consuming tasks in the effort to improve system quality. It takes 50% to 80% of the time used for verification depending on the level of automation of the verification tools.

State of the art verification tools aim to shorten the design cycle by decreasing the time necessary to detect a fault, but none of them address the issue of fault localization and correction. Thus, it is very important to provide new technologies to locate and correct faults in a fast and efficient way. **Property-based error localization** addresses exactly this issue. Property-based error localization is the process of isolating parts of the design likely to be responsible for the failure of the verification step. The design or verification engineer gets a small fraction of the circuit to

focus on during debugging, which speeds up the fault correction. Property-based error correction, which is an extension to property-based error localization, goes one step beyond fault localization and additionally provides a replacement for the faulty component. Fault localization and correction may **significantly reduce design cycle time by reducing debugging time**, which takes about 35% of the entire design cycle (see Figure 1).

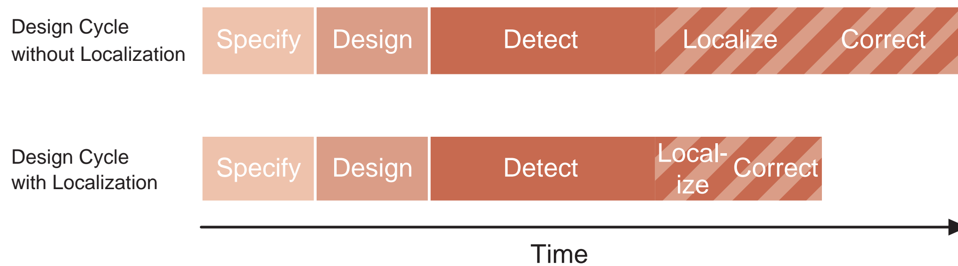

Figure 1: Time-saving by Fault Localization

Finding and fixing a fault in a fast and efficient way is great but it would be even better to avoid the fault in the first place. Automatically constructing the design from a given set of properties is the best way to avoid verification failure, because the constructed design is guaranteed to fulfill the properties. Given a set of properties that a design should fulfill, the aim of **property-based synthesis** is to automatically construct a design.

The major benefits of using a property-based synthesis tool are the following:

- We **avoid hand coding completely** for design blocks that are non-critical with respect to size and speed. The designs automatically generated are guaranteed to adhere to their functional specifications. Thus, we can shorten the design and the verification phase, together consuming about 85% of the design cycle time.
- Designer have the **ability to develop a design block in the presence of a fully operational environment**, because functional prototypes of the other blocks can be created right after specification. This allows integration testing to start right away and circumvents design mistakes. Automatically generated prototypes can also be used as a starting point for an efficient hand implementation.
- We can **detect specification faults very early** in the design process, as they are immediately apparent in the synthesized system. Thus, a correct specification is obtained earlier and incorrect implementations are avoided.

Therefore, property-based synthesis supports the development of systems with higher quality, within shorter time, and with lower costs.

Methodology Overview

Figure 2 shows an overview of the PROSYD tool set and the role of the property-based error localization and property synthesis tools in the design flow.

In the following, we will assume that the design is divided into blocks and that blocks are designed such that most of them are amenable to (formal) verification. This is achieved by using a small block size and well-defined interfaces between the blocks. (See Deliverable 3.1/1, "Combined Static and Dynamic Property Verification" [3].) The design of single blocks is supported by property-based error localization and property-based synthesis. Depending on the complexity of a block we can support the designer in different ways.

The methodology and benefits of error localization and property synthesis are described in the next two section. We describe them using design scenarios.

Error localization with or without correction. Section 2 describes the methodology and benefits of error localization. Error localization is described in PROSYD deliverable 2.2/2, "Property-based error localization" [24].

Property synthesis. Section 3 describes the methodology and benefits of property synthesis. Property synthesis is the subject of PROSYD deliverable 2.2/1 "Property-based logic synthesis for rapid design prototyping" [7].

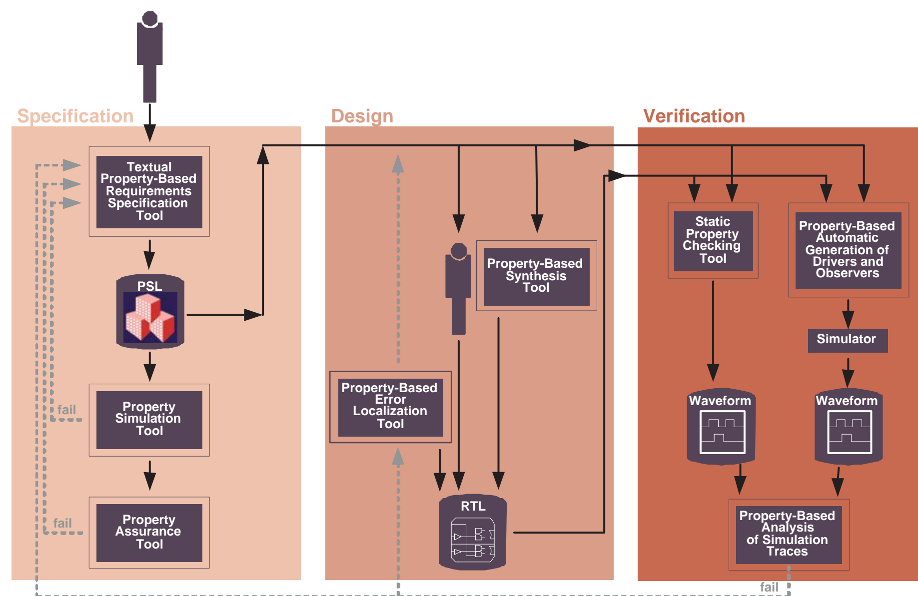

Figure 2: The PROSYD Tool Set

2 Error Localization

Background

The difficulty of deriving the cause of a failure from an example of incorrect behavior has been widely recognized. Different approaches have been taken to alleviate this problem. One possibility is to try to simplify the evidence of the incorrect behavior as much as possible. This has been the approach of Jin, Ravi, and Somenzi [12, 20] in the setting of formal verification, and of Zeller and Hildebrandt [29] in the setting of testing of software.

Another technique to help the user understand the incorrect behavior of the program is to look at multiple traces, some of which show the fault, and some of which do not. The similarities between incorrect traces, and their differences with similar correct traces give an indication of the parts of the program that are involved in the fault. Parts of the program that appear in many or all incorrect traces and in few or no correct ones are likely to be involved in the failure. This approach has, in different varieties, been attempted both in model checking [10, 2, 22, 8, 9] and in testing of programs [28].

Although they clear demonstrate the need to assist the user finding the fault, the approaches shown thus far explicitly target understanding of the failure, not fault localization. Our task is to localize the fault, that is, to limit the possible sources of the failure to a small portion of the program, which is of more help in correcting the fault. In some cases, our tools can even give a correct program.

There is ample work on locating faults in combinational circuits [16, 15, 4, 26]. The problem is considerably harder for sequential systems, unless a reference model with the same state space encoding is available. The only work in fault localization in sequential systems is that by Wahba and Borrione [27]. However, they deal with a very small set of possible errors (such as a forgotten inverter), and also need a reference model. We should stress that building a reference model is much more of a burden on the user than writing a set of properties. Wahba and Borrione's approach is useful mainly in the last stages of design, where, for example, an optimizer may make a mistake.

Our **fault localization** approach uses a finite set of traces, obtained from a dynamic or static verification tool, to limit the part of the system that may be at fault. It builds on model-based diagnosis [21] and extends it with the possibility to handle

sequential circuits and, more importantly, PSL properties. Such an approach has not been attempted before.

Fault correction builds on a synthesis technique and is the first approach to yield (nontrivial) corrections for a faulty circuits. Our very general fault model allows use of the tool in an early stage of design. The approach of Wahba and Borrione does yield correction, but only of the very limited sort mentioned. Furthermore, our approach bases decisions on properties, not on a reference model. Therefore, it can be used as soon as an a first implementation is available. The fault correction approach yields a circuit that is guaranteed to fulfill the specifications for all possible inputs.

Summarizing, our contribution is

1. Our approach uses PSL properties and does not need a reference model.
2. The fault correction approach finds a correction for the problem using a general fault model.
3. Our approach works for sequential circuits, not only for combinational ones.

Methodology and Examples

Property-based error localization can be performed with or without correction, and we will describe the two approaches separately. The **localization approach** uses failure traces obtained from a static or dynamic verification tool and performs fault localization without providing a fix. It is very efficient, is able to handle large and complex designs, and allows the user to focus her attention on a small part of the design. In the **correction approach**, which is more complex, fault localization and correction are combined. It takes the view that a component can only be responsible for the fault if it can be replaced by an alternative that makes the system correct. The approach is precise and provides the user with a correct replacement for the faulty components.

In the following, we describe the usage of the two approaches in more detail using concrete examples.

Localization

The localization scenario is shown in Figure 3. The input to the localization tool consist of

1. A buggy design that was constructed manually from the block-specification, and

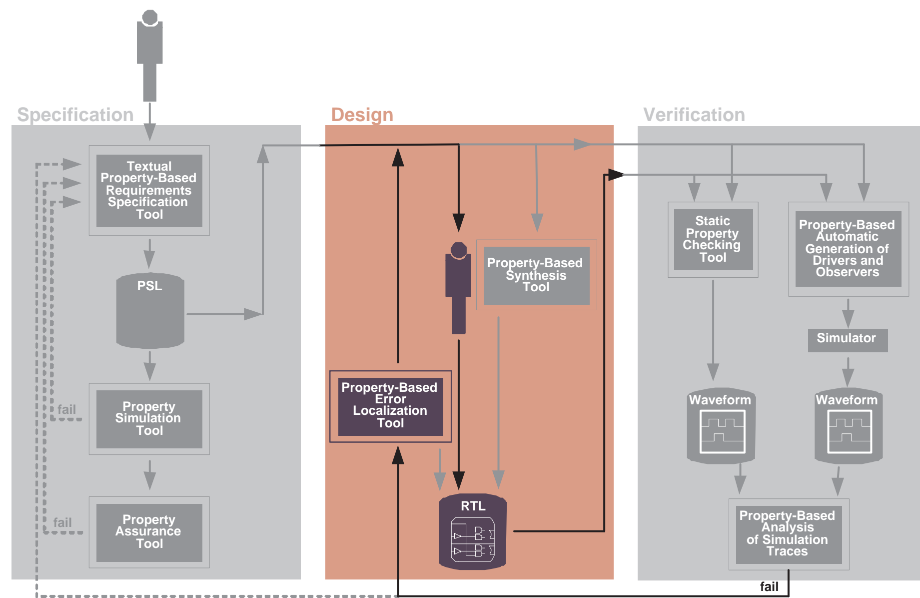

Figure 3: Fault Localization

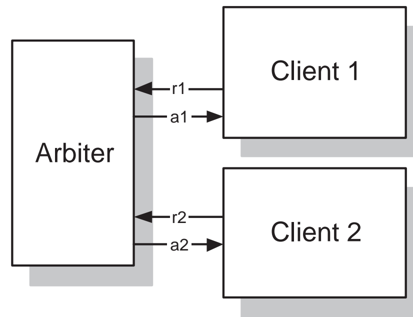

Figure 4: A Simple Arbiter

2. A set of counterexamples from the property-verification tool, showing how the design violates its specification.

The property-based error localization tool provides the user with a set of potentially faulty components. Thus, the user can concentrate on a small fraction of the design to find the source of the error, speeding up design correction significantly. Once the user has corrected the design it is verified again to check the manual correction.

Example

We show property-based error localization applied to the synchronous arbiter with two clients shown in Figure 4.

The arbiter has request lines r_i and acknowledgment lines a_i for $i \in \{1,2\}$. At any clock cycle a subset of the request lines are high. It is the task of the arbiter to set at most one of the corresponding acknowledgment lines high. The arbiter should be fair to all requests. In Table 1 we list the properties the arbiter should fulfill.

We manually construct the design shown in Figure 5. The three latches have the following meaning: Latch T toggles between 0 and 1 indicating which client gets

Table 1: Property Set of an Arbiter with two Clients

PSL Property	Meaning
never $(a_1 \wedge a_2)$	Mutual exclusion of acknowledgments
always ($\{r_1\}(a_1 \vee \text{next}(a_1))\wedge$ $\{r_2\}(a_2 \vee \text{next}(a_2)))$	Requests are granted within two ticks
$(r_1$ before_ $a_1)\wedge$ $(r_2$ before_ $a_2)$	No spurious acknowledgment at the beginning
always $(\{a_1\}(\text{next}(r_1$ before_ $a_1))\wedge$ $(\{a_2\}(\text{next}(r_2$ before_ $a_2)))$	No spurious acknowledgment

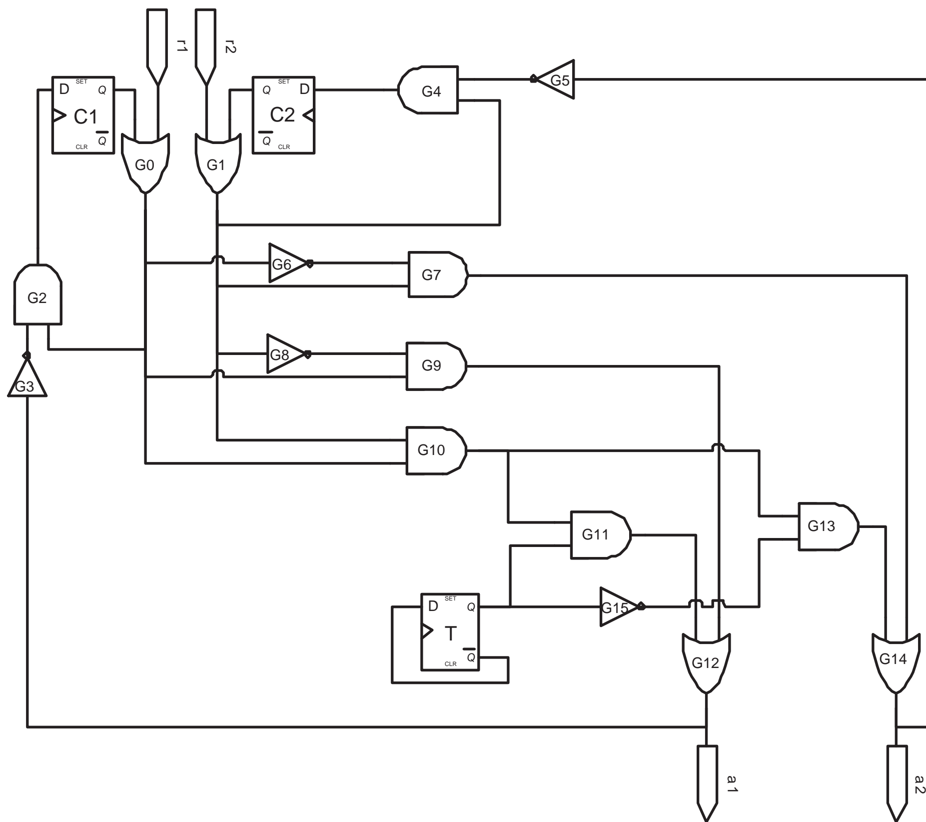

Figure 5: Manual Design

an acknowledge signal in this time step if both clients send a request. Latch C1 is high if a request of Client 1 has not be acknowledged yet. Latch C2 stores open requests of Client 2. Note that the output of G10 is high when a conflict occurs and that in this case, G11 and G12 decide which requester receives a grant.

Suppose that the designer incorrectly chooses an AND gate for G12, instead of an OR gate. Suppose furthermore that the verification tool finds the following error trace.

signal	time	
	t_0	t_1
r_1	1	0
r_2	1	0
a_1	0	0
a_2	1	0

This trace violates the second property because the request r_1 at time step t_0 is not acknowledged within two time steps. For this error trace, the property-based error localization tool computes that only gate G1, G8, G9, or G12 can be responsible for the fault. This reduces the amount of gates to be considered to under a quarter of the total.

Error localization provides the user with a small set of potentially faulty components that she can concentrate on during the debug process. This significantly reduces the design cycle time by reducing the time necessary to locate and correct a fault.

Applicability

Fault Localization can be applied to all blocks that are amenable to formal verification. There are no restrictions on the functional behavior of the block. In order to identify faulty components, the components of the system must be clearly defined. For instance, we can define the components at a low level (gates) or at a higher level (full adders, half adders, etc.). This decision is not critical to the implementation, and will be decided after experimentation and feedback from the case studies.

Correction

Apart from locating the fault, the property localization tool is able to suggest corrections for a faulty design. The corrections it proposes fulfill the properties for any possible input sequence, not just for a set of failure trace. The user then takes the suggested fault location and correction and adapts the design. Since there may be properties that are not stated explicitly, the user has to check that the correction adheres to the design intent. Figure 6 depicts the correction scenario.

Examples

Let us continue the arbiter example from the previous section. The fault localization tool excludes the possibility that a change to G1, G6, or G7 makes the circuit correct. The only possible change is to G12, for which the tool correctly suggests an OR gate.

A tool that automatically finds faults and suggests a correction has the potential to significantly reduce the time spend on debugging. As debugging takes one third of the design time, this may allow for much shorter design cycles. Besides, the suggested correction can help the user to get a better understanding of the design

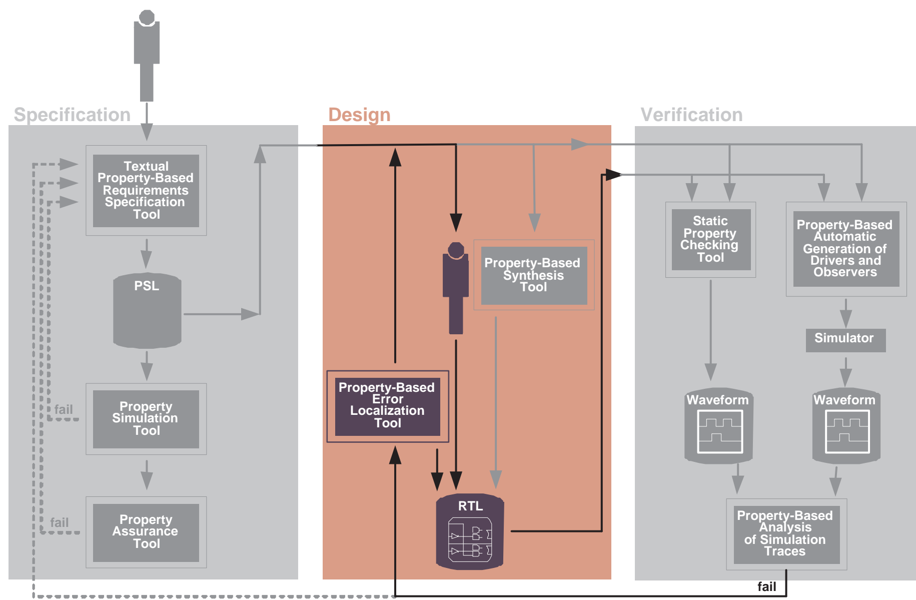

Figure 6: Fault Correction

and indicate possibilities for new design-optimizations. The following example shows these benefits.

We use the 8-Bit sequential multiplier depicted in Figure 7 to show how a design optimization can be found by the tool. The multiplier has two input shift-registers A and B, and a register Q that stores intermediate data. If input INIT is high, shift registers A and B are loaded with the inputs and Q is reset to zero. In every clock cycle register A is shifted right and register B is shifted left. The least significant bit (LSB) of register A is the control input for the multiplexer. If it is high, the multiplexer forwards the value of register B to the adder, which adds it to the intermediate result stored in register Q. After eight clock cycles register Q holds the product $A * B$.

Assume the multiplier has a fault in the adder: The output of the single-bit full adder responsible for Bit 0 (LSB) always adds 1 to the correct output. The components we use for fault localization are the eight full adders in the adder, the eight AND gates in the multiplexer, and the registers A, B, and Q. As the property we state that after eight clock cycles the output must be equal to $A * B$.

The property-based error localization tool finds the faulty part in the adder and provides a correction. It suggests to use a half adder for Bit 0, instead of the faulty component. We had expected a full adder, but the suggested repair is simpler. Using a half adder for Bit 0 ignores the carry bit of Bit 0. This is correct because the adder for Bit 0 never has to produce a carry bit. The LSBs of Q and B are never 1 at the same time because in the first time step, Q is 0 and in all subsequent steps, the LSB of B is 0 because B is shifted left. Consequently, a carry never occurs.

Thus, the correction tool can not only save time in debugging, but also help the designer gain a better understanding of the design, and even to find design optimizations.

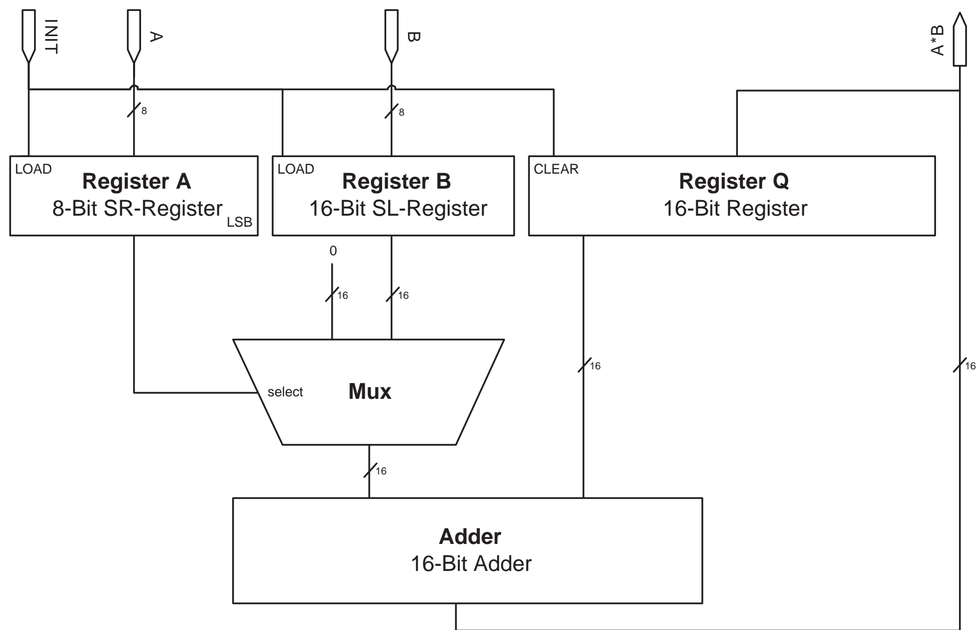

Figure 7: 8-Bit Multiplier

Applicability

In order to replace faulty components, the components of the system and their replacements must be clearly defined. As stated in the previous section the components of circuits are gates or sets of closely related gates. The replacements for gates in sequential circuits are Boolean functions in terms of states and inputs. Fault correction can be applied to all blocks applicable for formal verification, since we suggest a quite general fault model: we assume that any component can be replaced by an arbitrary function in terms of the inputs and the state of the system.

Tool

The property-based error localization tool implements the localization and the correction approach. In the following we discuss details of the tool.

Given a faulty design and a set of properties, static and dynamic verification tools check which properties are fulfilled and provide failure traces as evidence for the violated properties. The objective of the property-based error localization tool is to support finding and fixing faults by automating the localization and the correction of a fault.

Inputs and Outputs

The input to the error localization tool consists of a faulty circuit or system and evidence of the fault. The evidence is a set of failure traces obtained from a property verification tool. Additionally, the tool receives a set of PSL properties, at least one of which is violated by the system and corresponds to the failure traces. The output of the tool is a set of components that are likely to have caused the malfunctioning of the system. When we use the correction approach, the tool additionally provides correct replacements for the faulty components.

Components

The property-based error localization tool will contain the following components for the localization approach:

1. A parser that reads temporal assertions.
2. A model generator that constructs a combinational model based on the sequential design and the specification.
3. A converter that converts the combinational model into a CNF formula.
4. A SAT solver that find assignments to the CNF formula and therefore provides diagnoses.

The correction approach requires different components, but these can be shared with the property-based synthesis tool (see Section 3):

1. A parser that reads temporal assertions.
2. A compiler that constructs and interconnects state machines.
3. A strategy finder that search for an implementation.
4. A extractor that extract a correction from the strategy

Technical Approach

In the next two subsections we briefly describe the approaches we are going to use in the property-based error localization tool. For a detailed description see PROSYD deliverable 2.2/2 [24].

Localization

The localization approach is based on the theory of model based diagnosis, which aims to identify the cause of a fault by its effect. Model based diagnosis assumes that a correct model of the system is given and compares the behavior of the current (faulty) system with the model. Using the discrepancy between an observation of the faulty system and the correct model it identifies components likely to be responsible for the error. Model based diagnosis is either consistency or abduction based [6]. Abduction based diagnosis needs details on how a component may fail and tries to find a component and a corresponding fault that explains the observation. Consistency based diagnosis does not require knowledge of how components may fail. Rather, it tries to make the system consistent with the correct model by identifying a component such that dropping any assumptions on the behavior of the component removes the discrepancy. Consistency based diagnosis may be less precise than abduction based diagnosis but since it does not require information about the possible faults it is applicable more generally.

Our approach extends the ideas of consistency based diagnosis to sequential systems and PSL properties. Given a finite-state sequential system, a set of PSL-safety properties, and one or more error traces, we build a setting suitable for a variation of consistency based diagnosis. First, we unroll the sequential system to the length of the given error traces to get an equivalent combinational representation of the system. Then, from the PSL properties, we generate a model that recognizes the correct behavior of the system. To this end, we use expansion rules [17]. We combine this model with the unrolled sequential system and get a new combinational system. On this system we perform consistency based diagnosis to obtain all components that are likely involved in the failure.

Correction

The correction approach is based on the idea that a component can only be responsible for a fault if it can be replaced by an alternative that makes the system correct. Therefore, we combine fault localization and correction. Starting point is a faulty finite-state system and a specification given in PSL. We aim to find and fix a fault in such a way that the new system satisfies its specifications for all possible inputs. The fault model we consider is quite general: we assume that any component can be replaced by an arbitrary function in terms of the inputs and the state of the system.

Our approach is based on the work done in synthesis for PROSYD deliverable 2.2/1 [7], due 1 September 2005. Section 3 contains a brief overview of this work. (See also [13].) In that work, a method for the correction of a set of suspect components is presented. A restriction in the work is that a suspicion of the location of the fault has to be given by the user. We solve that restriction by combining fault localization and correction.

We consider the problem of fault localization and correction as an infinite game between two player, the system and the environment. The specification is given

as PSL properties and defines the winning condition for the system. The system tries to fulfill the specification whilst the environment tries to bar the system from winning. The two players can make different choices: The environment chooses the inputs values of the system at each time step. On the other hand the system has the freedom to slightly modify the implementation: At the beginning of each play it can pick a component for which it can choose arbitrary output values at each time step. The system wins the game if it can choose outputs for the selected component such that the specification is satisfied for any input sequence. Choosing the right values corresponds to finding a winning strategy for the game. If the corresponding strategy is memoryless, i.e., the output of the component depends only on the state of the system and its inputs, we can derive a replacement behavior for the component that makes the system correct. The method is complete for invariants, and in practice works well for general PSL properties, even though it is not complete.

3 Property Synthesis

Background

Automatic synthesis of a system from a set of properties has long been studied. Church [5] describes the problem for S1S. Pnueli and Rosner [18] describe the problem in the setting of Linear-Time Logic, which is similar to our setting. Their construction, however, requires the determinization of a Büchi automaton using Safra's determinization construction [23]. Safra's construction is generally considered to be very complex and not amenable to efficient implementation [25, 11]. Only recently have researchers found simpler methods to synthesize linear logic without resorting to Safra's construction [14], finally removing this important roadblock to an efficient implementation. An implementation of a synthesis tool for LTL, PSL, or a similar logic does not currently exist, and considerable research is still required to turn the theoretical results into a practical algorithm.

Methodology and Examples

Given a set of properties for a block, the property-based synthesis tools provides a functionally correct design of the block. Figure 8 shows the integration of the tool in the general design flow: Once a block is specified, the PSL properties are passed on to the synthesis tool. The synthesis tool generates a valid design. In the verification phase, the generated design is used for integrating testing and the verification of other blocks. For blocks non-critical with respect to size or speed, there is no need for further optimization. The design provided by the synthesis tool is the final design.

Examples

Suppose we want an automatically synthesized design for an arbiter that fulfills the properties listed in Table 1. Since the arbiter will works over a slow bus, it has no

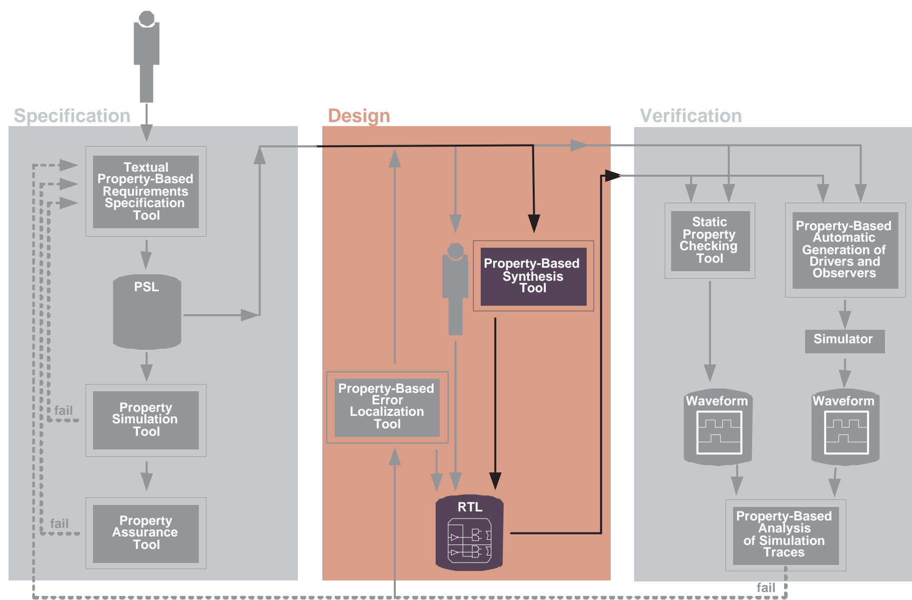

Figure 8: Synthesis

timing constraints. We feed the properties into the properties synthesis tool, and synthesize them. The tool yields a design like the one in Figure 9. Since there is no need for a faster or smaller design, we have our final design. So we can **avoid hand coding completely**. The properties are guaranteed to be fulfilled. Note that verification may still be necessary for integration testing and to make sure that the specification does not contain any mistakes.

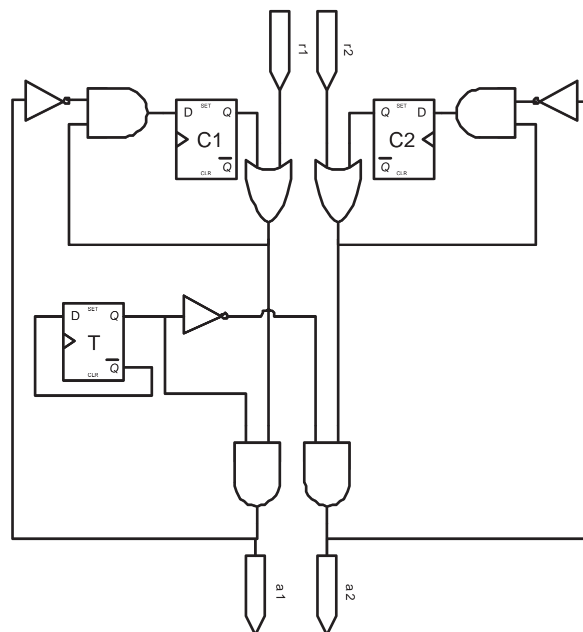

Figure 9: Simple Design

Let us assume we get additional constraints concerning the clock speed of the arbiter. We generate a first prototype using the synthesis tool. Machine-generated hardware is generally not as efficient as human-written hardware. So suppose the generated design is more complex than the one in Figure 9 and needs a slow clock

frequency. Now we want to improve the **design manually, using the generated prototype**. We set up an **executable environment** to verify our optimization step immediately. Figure 10 shows an example environment: The block in the middle represents the arbiter we want to optimize. The blocks to the left and above are the two client the arbiter schedules. On the right side we find the device the clients want to access. Using the synthesis tool, we can generate this executable environment from the properties for the arbiter and the clients, without having manual implementations.

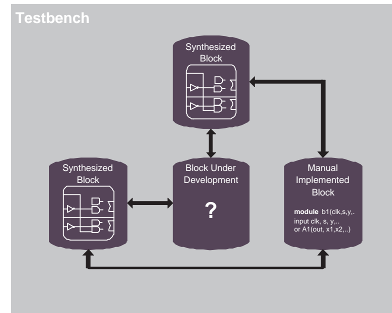

Figure 10: Synthesis of Blocks in Executable Environment

The setting in Figure 10 also allows designers to implement blocks depending on each other simultaneously. For instance, each client of the arbiter is developed by one designer. Both designers will be able to test their blocks without waiting for the other blocks to be finished. When system integration and testing begins, the blocks are thoroughly verified and are likely to work together because they were developed to satisfy the common set of properties.

A early synthesized system helps to **detect specification faults very soon** in the design process because they are immediately apparent. For instance, suppose we want a fair arbiter for two clients. Assume we are given an incomplete specification consisting of only the first two properties of Table 1, stating that the arbiter should guarantee mutual exclusion and be fair to all requests. The synthesis tool provides a design like the one shown in Figure 11, which fulfills the given properties. We would not be satisfied with an arbiter implementation that alternatively sets a_1 and a_2 to high. However, one simple simulation run instantly shows that we did not get what we expected. Thus, the designer writing properties of this specific block immediately gets a feedback about what she specified.

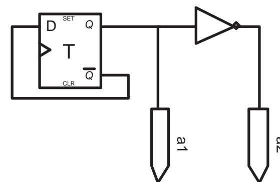

Figure 11: Simple design for an incomplete property set

Another advantage of property-based synthesis is the ability to help generate new design ideas. First experimental results have shown that automatically generated design sometimes are quite different from designs someone expects. Figure 11

and Figure 9 show two unexpected implementations for the arbiter introduced at the beginning of this section. The design in Figure 9 is especially interesting, since it is smaller than our manual implementation in Figure 5 and still has the functionally we intended.

Applicability

In general, property-based synthesis is applicable to designs for which speed and size are not critical. Therefore, synthesis is suitable for systems that focus on behavioral aspects, e.g., controllers, mobile devices, other slow-clock systems, and FPGAs.

A risk of property-based synthesis is that realization of temporal-logic formulas is known to be a high-complexity problem. This problem is alleviated by the fact that complex designs are always split into small blocks and we apply property synthesis on the block level. Additionally, we will work on optimization methods to handle the complexity problem. There is another way to deal with the complexity challenge within the scope of the PROSYD project. Based on deliverable 1.5/1 [19], which can provide ready-to-use property libraries for designated hardware cores and protocols, several blocks can be synthesized in advance. Specification properties and test implementation could come in pairs ready-to-use for formal verification and simulation.

Even though designs generated manually are generally better optimized than automatically generated ones, property-based synthesis is useful to get a rapid prototype for implementing blocks that are critical in terms of size and speed. The final implementation can be the result of a manual optimization of this prototype. Furthermore, rapid prototypes are useful for early integration testing.

Tool

The property-based synthesis tool is concerned with the construction of a gate-level or RTL hardware design from a set of PSL properties. The construction has to ensure that this piece of hardware satisfies all given properties. The tool focuses on the behavioral description of the circuit, without paying attention to timing and area concerns.

Inputs and Outputs

The principal input to a property-based synthesis tool is a set of properties written in PSL [1]. PSL assertions refer to the temporal behavior of a clock-based hardware model. For instance, “if signal r is high eventually signal a gets high too”, or

“signal c is high always implies that c is low in the next time step”. Apart from that, the tool receives a description stating which signals are inputs and which are outputs. Outputs are controllable signal. The input signals are not controllable by the design.

In case of contradictory requirements the tool will state that the given set of properties cannot be synthesized. An example of contradictory requirements is “after input a has been high, output b will be high for ever and after input c has been seen, output b will be low forever”. Such a specification can only be synthesized under the assumption that signals a and c never assert in the same trace.

The output of the property-synthesis tool is synthesizable code written in a hardware description language such as Verilog or VHDL. The output is processed and optimized by a silicon compiler.

Components

The property-based synthesis tool will consist of the following four components.

1. A parser that reads temporal assertions.
2. A compiler that constructs and interconnects state machines.
3. A strategy finder that search for an implementation.
4. A extractor that translated the strategy into a design

Technical Approach

In this section we briefly describe the idea on which the property-based synthesis tool will be based. PROSYD deliverable 2.2/1 [7] (due 1 September 2005) provides a detailed description.

Assume we are given a set of properties written in PSL and a partitioning of the atomic proposition into input and output signals. Synthesizing these properties can be seen as an infinite game between two players. One player is the environment of the system and provides the inputs. The other player is the system and defines the output values. The system wins the game if it can choose output values such that it fulfills the given properties for all possible inputs sequences. A winning strategy for the system is a recipe with which the system is guaranteed to win the game no matter what input values the environment chooses. If a winning strategy exists, it prescribes the behavior of the system for any given input. This behavior is then turned in to a hardware description.

To find a winning strategy we will use recently achieved results in the theory on infinite word and tree automata [14]. Technically, given a PSL formula φ and

a signal partitioning (I, O), first we construct a nondeterministic Büchi word automaton $A_{\neg\varphi}$ accepting all words over $2^{I \times O}$ satisfying $\neg\varphi$. Using a special inverting technique, we obtain a universal co-Büchi tree automaton A_φ that accepts a 2^O-labeled 2^I-tree if it fulfills φ. A_φ is turned into a nondeterministic Büchi tree automaton on which we can compute language emptiness. If the language for φ is not empty we get a witness for φ. The witness is a 2^O-labeled 2^I-tree, which corresponds to a winning strategy $f : (2^I) \rightarrow (2^O)$.

We will also address the problem of automatically correcting faults in deliverable 2.2/1 because it is closely related to the synthesis problem. Contrary to the synthesis problem, we only need as much of the specification as is necessary to choose a correct replacement for the correction problem. Using the same ideas as above we can solve this problem as well. We consider the correction problem for finite state systems again as a game. The game consists of the product of a modified version of the system and an automaton representing the PSL specification. Every winning strategy for the system corresponds to a correction. Since we aim for corrections that yield to simple modified systems, which makes it amenable to further modification by the user, we will introduce heuristics to get a simple correction.

4 Conclusions

We have described the two areas of research in Workpackage 2: property-based fault localization and property synthesis. We have described what property-based fault localization and property synthesis are, what their place in the design cycle is, and their advantages and disadvantages. We have also given a short overview of the components of the tools and the technical approach.

Error localization addresses the debugging that has to take place when a verification tool finds a failure in a design. In traditional design flows, debugging takes 35% of total design time. In a property-based design flow it takes even more, as detection of failures is faster. Thus, there is a significant potential for saving time. Error localization comes in two variants. The localization approach receives faulty traces from a verification tool and returns a small subset of the design that contains the fault, thus allowing the user to focus her attention on the proper portion of the code. The correction approach yields the location of the fault and an alternative implementation that satisfies the specification. It is much more powerful, but less efficient than the localization approach.

Property synthesis addresses the problem of automatically turning a specification into a design. It replaces the traditional design flow and most of the verification flow. Verification is still necessary to test if the specification adheres to the design intent. It has three major advantages: 1) Hand coding and most of verification is avoided for blocks that are not timing or area critical, 2) hand design of critical blocks can start from a prototype, and can take place in an executable environment, enabling immediate integration testing, and 3) specification faults are found immediately and a correct specification is available sooner.

5 References

[1] PSL/Sugar consortium. Homepage http://www.pslsugar.org/.

[2] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *30th Symposium on Principles of Programming Languages (POPL 2003)*, pages 97–105, January 2003.

[3] Jörg Bormann, Andrea Fedeli, Roy Frank, and Klaus Winkelmann. Combined static and dynamic verification. Prosyd D 3.1/1.

[4] P.-Y. Chung, Y.-M. Wang, and I. N. Hajj. Logic design error diagnosis and correction. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2:320–332, 1994.

[5] A. Church. Logic, arithmetic and automata. In *Proceedings International Mathematical Congress*, 1962.

[6] L. Console and P. Torasso. A spectrum of logical definitions of model-based diagnosis. *Computational Intelligence*, 7(3):133–141, 1991.

[7] Prosyd TU Graz. Property-based logic synthesis for rapid design prototyping. Prosyd D 2.2/1 due on 1st September 2005.

[8] A. Groce. Error explanation with distance metrics. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'04)*, pages 108–122, Barcelona, Spain, March-April 2004. LNCS 2988.

[9] A. Groce, D. Kroening, and F. Lerda. Understanding counterexamples with `explain`. In R. Alur and D. Peled, editors, *Sixteenth Conference on Computer Aided Verification (CAV'04)*, pages 453–456. Springer-Verlag, Berlin, July 2004. LNCS 3114.

[10] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *Model Checking of Software: 10th International SPIN Workshop*, pages 121–135. Springer-Verlag, May 2003. LNCS 2648.

[11] S. Gurumurthy, O. Kupferman, F. Somenzi, and M. Y. Vardi. On complementing nondeterministic Büchi automata. In *Correct Hardware Design and Verification Methods (CHARME'03)*, pages 96–110, Berlin, October 2003. Springer-Verlag. LNCS 2860.

[12] H. Jin, K. Ravi, and F. Somenzi. Fate and free will in error traces. *Software Tools for Technology Transfer*, 6(2):102–116, August 2004.

[13] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. To appear at Computer Aided Verification '05, retrieve from www.ist.tugraz.at/verify/view/Projects/ProgramRepair, 2005.

[14] O. Kupferman and M. Vardi. Safraless decision procedures. Unpublished.

[15] H.-T. Liaw, J.-H. Tsiah, and I. N. Hajj. Efficient automatic diagnosis of digital circuits. In *Proceedings of the International Conference on Computer-Aided Design*, pages 464–467, 1990.

[16] J. C. Madre, O. Coudert, and J. P. Billon. Automating the diagmosis and the rectification of design error with PRIAM. In *Proceedings of the International Conference on Computer-Aided Design*, pages 30–33, 1989.

[17] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems *Specification**. Springer-Verlag, 1991.

[18] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. Symposium on Principles of Programming Languages (POPL)*, pages 179–190, 1989.

[19] Prosyd. Manual for repositories of psl/sugar properties. Prosyd D 1.5/1 due on the end of 2006.

[20] K. Ravi and F. Somenzi. Minimal assignments for bounded model checking. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'04)*, pages 31–45, Barcelona, Spain, March-April 2004. LNCS 2988.

[21] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.

[22] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *International Conference on Automated Software Engineering*, pages 30–39, Montreal, Canada, October 2003.

[23] S. Safra. On the complexity of ω-automata. In *Symposium on Foundations of Computer Science*, pages 319–327, October 1988.

[24] S. Staber and R. Bloem. Property-based fault localization. Prosyd D 2.2/2 due on 1st May 2005.

[25] S. Tasiran, R. Hojati, and R. K. Brayton. Language containment using non-deterministic omega-automata. In *Correct Hardware Design and Verification Methods (CHARME95)*, pages 261–277, Berlin, 1995. Springer-Verlag. LNCS 987.

[26] M. Tomita, T. Yamamoto, F. Sumikawa, and K. Hirano. Rectification of multiple logic design errors in multiple output circuits. In *Proceedings of the Design Automation Conference*, pages 212–217, 1994.

[27] A. Wahba and D. Borrione. Design error diagnosis in sequential circuits. In *Correct Hardware Design and Verification Methods (CHARME'95)*, pages 171–188, 1995.

[28] A. Zeller. Isolating cause-effect chains from computer programs. In *10th International Symposium on the Foundations of Software Engineering (FSE-10)*, pages 1–10, November 2002.

[29] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.