Fehlerinjektionstechniken in SytemC-Beschreibungen mit Gate- und Switch-Level-Verhalten

Silvio Misera, André Sieber
BTU Cottbus, Lehrstuhl Technische Informatik
<sm, asieber@informatik.tu-cottbus.de>

Kurzfassung

Zur Beschreibung elektronischer Systeme hat SystemC inzwischen eine festen Platz in der Entwurfslandschaft gefunden. Ein wesentlicher Vorteil eines SystemC-Modells ist die bereits vorhandene Möglichkeit einer Simulation. Neben der rein funktionalen Simulation zur Entwurfsvalidierung ergeben sich für eine Simulation mit injizierten Fehlern zusätzliche Herausforderungen. In dieser Arbeit werden diverse Techniken zur Fehlerinjektion in SystemC vorgestellt. Einige vergleichende Experimente helfen diese Techniken zu bewerten. Anschließend werden einige Modelle präsentiert, die es gestatten, SystemC auch auf niederen Ebenen des Hardwareentwurfs einzusetzen. Mit den vorgeschlagenen Methoden eröffnet sich hiermit die Möglichkeit einer genauen Untersuchung zur Auswirkung von Hardwarefehlern in digitalen Schaltungen mit Hilfe von SystemC.

1. Einführung

Der Entwurf elektronischer Systeme steht seit Jahrzehnten immer wieder vor der Herausforderung, dass zunehmende Integrationsdichten und wachsende Dimensionen neue Entwicklungsstrategien erfordern. Der übliche, komplexe Entwurf findet heute nach der Spezifikation auf der Systemebene seinen Einstieg. Zur Beschreibung hat sich hier SystemC [8] als ein Quasi-Standard innerhalb kurzer Zeit etabliert. Mit dem Erscheinen von SystemC als eine Erweiterung von C++ in Form einer Klassenbibliothek, hatte man zunächst die Absicht den Umstieg auf eine Hardwarebeschreibungssprache (HDL), der sich zuvor im Laufe der Entwurfsverfeinerung ergab, mit einen durchgehenden Flow zu vermeiden. Außerdem erhoffte man sich durch die Nutzung einer hohen Programmiersprache ein einfacheres HW/SW-Codesign. Beides sieht heute etwas anders aus. So findet ein vollständiges HW/SW-Codesign nicht komplett in einer SystemC-Umgebung statt. Problematisch ist hier beispielsweise eine nicht taktzyklengenaue Nachbildung der Instruktionen des Zielprozessors in der Entwicklungsumgebung. Ebenso stehen Real Time Operating Systems (RTOS) [9] oder Instruktion Set Simulatoren nicht ohne weiteres bereit. Desgleichen führen die meisten SystemC-Modellierungen meist nur bis zum Register-Transfer-Level hinab. Entweder erfolgt von dieser Ebene schon die HW-Synthese oder es kommt zu einem Umstieg auf eine HDL wie z.B. Verilog oder VHDL. Das liegt nicht selten daran, dass HW-Entwickler auf den unteren Ebenen des Systementwurfs oft eher mit einer HW-Beschreibungssprache vertraut sind als mit einer höheren Programmiersprache. Dennoch wäre es sinnvoll, auch unterhalb des RT-Levels SystemC-Darstellungen zu benutzen. Zum Einen wäre so eine Multi-Level-Simulation deutlich einfacher bzw. die Simulation deutlich schneller. Zum Anderen lässt sich dadurch ein Äquivalenzvergleich, welcher bei einem Verfeinerungsschritt sinnvoll ist, besser durchführen.

Moderne Chips werden durch beständig kleiner werdende Strukturen immer störanfälliger gegen äußere Einflüsse wie elektromagnetische Strahlung oder Partikelstrahlung. Diese äußern sich typischerweise in sporadisch auftretenden, transienten Fehler. Um das Systemverhalten bezüglich der Auswirkungen solcher transienter Fehler näher zu untersuchen, ist die Methode der simulierten Fehlerinjektion eine beliebte Form. Für VHDL-Beschreibungen gilt dieses Gebiet als gut untersucht. In SystemC sind hierzu wenige Arbeiten bekannt und zumeist bezüglich der Modellierungsebene oder der Fehlerinjektionstechnik beschränkt.

In der folgenden Arbeit werden verschiedene Fehlerinjektionstechniken vorgestellt, wie sie zum Teil in ähnlicher Form schon aus früheren Arbeiten mit VHDL-Modellen bekannt sind. Ergänzend werden einige Simulationsergebnisse anhand von Benchmark-

Schaltungen präsentiert. Für eine genaue Untersuchung einer Fehlererscheinung reicht die Darstellung auf dem Register-Transfer-Level oftmals nicht. Ein feineres Modell auf dem switch-level spiegelt hierfür die Realität besser wider und erlaubt so genauere Fehlermodelle.

2. Techniken der Fehlerinjektion

Das Verfahren der simulierten Fehlerinjektion in Hardware-Beschreibungen ist ein recht gut untersuchtes Gebiet. Hierzu gibt es speziell viele Arbeiten für VDHL [1,2,3,4]. Abbildung 1 zeigt in Anlehnung an [3] eine Systematik verschiedener Fehlerinjektionen für HW-Beschreibungssprachen. Die in der Abbildung verwendeten Bezeichnungen sind gemeinhin in der Fachwelt als Quasi-Standard akzeptiert. Abweichende Begriffe in anderen Arbeiten finden nahezu immer ein entsprechendes Äquivalent zu den vorgestellten Methoden.

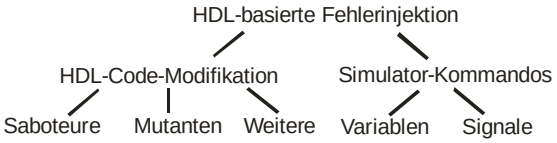

Abbildung 1: Eine Systematik von Fehlerinjektionen in HDL-Beschreibungen

Hinsichtlich der Fehlerinjektion sind die drei Techniken der Saboteure (saboteurs), Mutanten (mutants) und Simulator-Kommandos (simulator commands) von wesentlicher Bedeutung. In SystemC gibt auch einige, wenige Publikationen zur simulierten Fehlerinjektion [5,6,7]. Typisch für die Arbeiten in SystemC ist, dass ihr Fokus auf eine spezielle Anwendung ausgerichtet ist und das sie nur einzelne Fehlerinjektionen nutzen. Ein umfassende Implementierung und Untersuchung verschiedener Fehlerinjektionen findet man nicht.

2.1 Saboteure

Saboteure sind spezielle Module die der Originalbeschreibung hinzugefügt werden. Für gewöhnlich werden sie in den Signalweg von mindestens zwei Komponenten geschalten. Über einen zusätzlichen Kontrolleingang kann man im Saboteur die Fehlerinjektion steuern. Der Ablauf sieht für gewöhnlich so aus, dass während der Simulation ein Fehler aktiviert wird und man diesen nach einer festgelegten Zeit wieder deaktiviert.

Bei den Saboteuren unterscheidet man drei Kategorien:
Der *einfache serielle Saboteur* wird zwischen zwei Ports geschalten. Mit ihm lassen sich alle Fehler injizieren, die für eine Signalleitung denkbar sind. Das können z.B. Stuck-At-, Bit-Flip-, Stuck-Open- oder Verzögerungsfehler sein.
Der *komplex serielle Saboteur* wird zwischen mehreren Signalleitungen geschalten. Mit ihm lassen sich z.B. Brückenfehler und Übersprechverhalten modellieren. Aber auch als kompaktes Injektionsmodul für Einzelfehler auf Bussen ist dieser Saboteur geeignet.
Der *parallele Saboteur* wird an mindestens eine bestehende Leitung angeschlossen. Mit ihm lassen sich Störungen mit transientem Verhalten auf Bussen gut provozieren.
Die Abbildung 2 zeigt den Code eines einfach seriellen Saboteurs bei dem ein Teil der Funktionalität zu sehen ist.

```
   . . .
2  switch (fi.read()) {
     case 0: out.write(in.read());
4            break;        //correct
     case 1: out.write(sc_logic_0);
6            break;        //stuck-at-0
     case 2: out.write(sc_logic_1);
8            break;        //stuck-at-1
     case 3: out.write(~(in.read()));
10           break;        //bit-flip
     case 4: next_trigger(t_delay);
12           out.write(in.read());
             break;        //delay
14 . . .
```

Abbildung 2: Einfach serieller Saboteur

2.2 Mutanten

Mutanten führen zu einer Änderung der Schaltungsbeschreibung. In Hardware-beschreibungen wird üblicherweise eine Komponente durch eine andere ersetzt. Dieser Austausch kann z.B. auf zwei Wegen erfolgen. Im einfachsten Fall wird eine korrekt arbeitende Komponente gegen eine fehlerhafte mit anderer Funktion getauscht. Zum Beispiel wird ein AND-Gatter durch ein NAND-Gatter ersetzt. In SystemC würde man hierzu den Programmcode ändern, anschließend Compilieren und danach ausführen. Für die Injektion mehrerer, sequentieller Fehler ist dieses Verfahren nicht praktikabel, da die Zeit der Simulations-vorbereitung erheblich anwächst.
Die Alternative ist, die korrekte Komponente durch ein erweitertes Modul auszutauschen, welches die zusätzliche Möglichkeit der Fehlerinjektion bietet. Mit einem zusätzlichen

Kontrollport lässt sich dann der Fehlerzustand selektiv ein- und ausschalten.
Dieses Methode zeichnet sich durch eine hohe Flexibilität aus. Es lassen sich sowohl Signale als auch Variablen beeinflussen. Mit Mutanten sind alle Fehlermodelle möglich, die in einer Komponente entstehen können.
Die Abbildung 3 zeigt den Code eines Mutanten bei dem ein Teil der Funktionalität wieder gegeben ist.

```
   . . .
2  switch (v_fi) {
     case 0:
4      z0.write(a0.read()&a1.read());
       break;              //correct
6    case 1:z0.write(sc_logic_0);
            break;         //stuck-at-0
8    case 2:z0.write(sc_logic_1);
            break;         //stuck-at-1
10   case 3:next_trigger(t_delay);
       z0.write(A0.read()&A1.read());
12     break;              //delay
     case 4:
14    z0.write(~(A0.read()&A1.read()));
      break;               //NAND
16 . . .
```

Abbildung 3: Mutant in SystemC

2.3 Simulations-Kommandos

In einigen leistungsfähigen HDL-Simulatoren [11,10] gibt es eine Schnittstelle, die es dem Anwender mit speziellen Simulator-Kommandos ermöglicht, den Ablauf interaktiv zu steuern. So sind die Werte von Signalen und u.U. auch die von Variablen manipulierbar. Mit der Interpretation von Skripten lässt sich eine Simulation zusätzlich gut automatisieren. Ein Vorteil dieser Technik ist es, dass für die Fehlerinjektionen keine Modifikationen am Schaltungsmodell nötig sind. In [1] wurde anhand von VHDL-Modellen gezeigt, dass die zusätzliche Zeit zur Fehlerinjektion im Vergleich zu einer funktionalen Simulation nicht ins Gewicht fällt. Die Möglichkeiten der Fehlermodellierung bleiben jedoch auf die gebotenen Befehle des Simulators beschränkt. Etwas kompliziertere Fehler sind nicht modellierbar. In der SystemC-Bibliothek sind solche Kommandos nicht zu finden. Es ist zwar möglich während des Ablaufs aus der Simulationssteuerung heraus einem Signal oder einer Variablen einen Wert zuzuweisen, jedoch ist diese Injektion meist nur vorübergehend und kann beim nächsten Schreibvorgang wieder unwirksam sein. Ein solcher Vorgang lässt sich mit nur einer Anweisung erreichen. Für eine Fehlerinjektion in ein Signal genügt: *Modulname.Signalbezeichnung.write(Wert)*.

Leider ist diese Methode ungeeignet um permanente Fehler oder Fehler festgelegter Zeitdauer zu provozieren. Im Folgenden werden einige Methoden vorgestellt, die dieses Verhalten nachzubilden helfen. Mit den Simulator-Kommandos haben sie gemeinsam, dass sie ähnlich leicht zu handhaben sind. Jedoch unterscheiden sie sich in der Art ihrer Implementierung. Deshalb werden sie in Anlehnung an den Simulator-Kommandos als *Simulations-Kommandos* bezeichnet.

Für die *Logik-Signale* wurde das Problem so gelöst, dass in der SystemC-Bibliothek einige Klassen überschrieben wurden. Mit diesen mehrwertigen Logiksignalen lässt sich Hardware-Verhalten am genauesten simulieren. Folgende Änderungen waren dabei von Bedeutung:

- Der Datentyp *sc_logic* wurden um die zusätzlichen Datenwerte (´B´, ´A´, ´R´) erweitert. *sc_logic* $\in$ *{0, 1, Z, X, A, B, R}*
- Diesen Werten wurden in der Klasse *sc_signal_resolved* neue Signalstärken zugewiesen. Der Wert ´A´ (Above) bedeutet eine starke ´1´, der Wert ´B´ (Below) entspricht einer starken ´0´ und der Wert ´R´ (Release) ist ein neuer Sondertyp.
- Damit die neuen Werte auch sinnvoll ausgeführt werden können, wurden die Operatoren von *sc_logic* $\in$ (&, |, ^, ~, etc.) überladen.
- Zusätzlich wurde die *conversion table* für eine brauchbare Typumwandlung erweitert.
- Andere Methoden, wie sie z.B. für das Signal-Tracing nützlich sind, wurden überschrieben.

Mit den neuen Signalwerten eröffnet sich der Weg zur Injektion von Haftfehlern. Das Schreiben von ´A´ führt zu einem Stuck-At-1-Fehler und ´B´ zu einem Stuck-At-0-Fehler. Das Anlegen eines ´R´ hebt die Fehlerinjektion wieder auf. Die Tabelle 1 zeigt die Auflösung der Signalstärken.

	0	**1**	**Z**	**X**	**B**	**A**	**R**
0	0	X	0	X	B	A	0
1	X	1	1	X	B	A	1
Z	0	1	Z	X	B	A	Z
X	X	X	X	X	B	A	X
B	B	B	B	B	B	X	R
A	A	A	A	A	X	A	R
R	0	1	Z	X	R	R	R

Tabelle 1: Auflösung der Simulations-Kommandos

Bei näherer Betrachtung fällt auf, dass die Werte ´A´ und ´B´ ein dominantes Verhalten besitzen. Allerdings ist ein ´R´ stärker als ´A´ und ´B´. Andererseits ist ´R´ schwächer als logisch ´0´ und ´1´ . Mit diesem Trick wird die Fehlerinjektion wieder aufgehoben.
Um während der Simulation Fehler aktivieren zu können und sie auch wieder aufzulösen, kam es in der *main*-Datei zur Implementierung einer Simulationssteuerung. Diese sieht den Ersatz der Standard-Anweisung *sc_start(..)* durch die Befehle *sc_initialize( ), sc_cycle( )* und eine direkte Beeinflussung des Taktes vor. Eine Fehlerinjektion kann zwischen *sc_cycle( )*-Anweisungen ihren Platz finden. Für einen Stuck-At-1-Fehler kann z.B. das Kommando wie folgt aussehen:
`Mod1.SubModA.Sig1.write(sc_logic_A).`

Für die Injektion von permanenten Fehlern bzw. solcher mit deterministischer Zeitlänge in *Variablen* ist die für Logik-Signale vorgeschlagene Methode problematisch. Zum Einen ist der Wertebereich einer Variablen für gewöhnlich weitaus größer bzw. ausgeschöpft und zum Anderen erlaubt C++ keinen Eingriff in seine proprietären Datentypen. Um dieses Problem zu lösen, wurde eigens ein neuer Datentyp erstellt. Der neue Datentyp verhält sich so, dass er in Abhängigkeit eines Zugriffs-Flags, einen Schreibvorgang erlaubt oder unterbindet. Die Abbildung 4 zeigt einen Ausschnitt der Klasse *MyInt* die als Ersatz zum Integer-Datentyp dient.

```
   . . .
2  class MyInt {
     private:
4       int dat;
        bool lock;
6    public:
       MyInt (int dat_=0) {
8           this->dat =  dat_;
            lock = false;
10      }
   . . .
12    inline MyInt operator &=
        ( const MyInt& b ) {
14          dat = dat & b.dat;
            return this->dat;
16    }
      inline bool setlock() {
18          this->lock=true;
            return true;
20 . . .
```

Abbildung 4: Fehlerinjektionsdatentyp für Integer

Member-Variablen des neuen Datentyps sind *dat* für die Aufnahme des Datums und *lock* für den Zugriffsschutz. Um mit diesen Datentypen vernünftig arbeiten zu können, wurden alle nötigen Operatoren [12] überladen (z.B. Zeile 12-16). Ein Operator beeinflusst dabei nur die *dat*-Variable. So verhält sich eine *MyInt*-Variable funktional nach außen wie eine normale Integer-Variable. Für den Zugriff auf das *lock*-Flag wurden separate Methoden erstellt (z.B. Zeile 17-19). Um einen definierten Fehlerwert zu setzen, geht man z.B. so vor, dass zunächst der fehlerhafte Wert geschrieben und anschließend die Variable mit *setlock( )* gesperrt wird. Für ein feineres Modell, z.B. eines Registers, kann es nötig sein, nur einzelne Bitstellen auszuwählen. Hierzu wird die vorgestellte Klasse zur Bit-Selektion um eine Masken-Member-Variable erweitert. Für ein fertiges Fehlerinjektionsprojekt muss dann lediglich in der HW-Beschreibung der bestehende Datentyp durch den erweiterten Typ ersetzt werden. Die Einführung der Simulations-Kommandos für Variablen erlaubt eine präzisere und erweiterte Fehlerinjektion.

2.2 Vergleich der Simulationstechniken

Hauptkriterium für die Wahl einer bestimmtem Fehlerinjektionstechnik wird der zu modellierende Fehler sein. So sind Saboteure für Fehler zwischen Komponenten und Mutanten für Fehler in Komponenten prädestiniert. Es gibt aber auch Fehler die sich in allen drei Techniken modellieren lassen. Insbesondere trifft dies auf das klassische Haftfehlermodell zu. Weiteres Kriterium ist der Aufwand der in die Simulationsvorbereitung gesteckt wird. In unserer Arbeit gelang es diesen Aufwand mit Hilfe von Skripten [13] weitgehend zu automatisieren. Natürlich sind auch mehrere Fehlerinjektionstechniken in einem Modell verwendbar. Einen Schritt weiter geht der Ansatz, dass Saboteure und Mutanten mit den Simulations-Kommandos gemischt werden. Ziel dieser Idee ist es, die zusätzlichen Ports und Verbindungen für die Aktivierung eines Fehlers durch Kommandos zu ersetzen.
Die Tabelle 2 zeigt die Simulationsergebnisse für zwei Schaltungen und den verschiedenen Techniken. Wie sich unschwer erkennen lässt, ist die Technik der Simulations-Kommandos die Schnellste. Ebenso hat die Mischung von Mutant und Kommandos Geschwindigkeits-vorteile gegenüber der reinen Mutant-Technik. Gleiches gilt für die Saboteure. In der letzten Zeile sind Ergebnisse mit FIT dargestellt, einem Fehlerinjektionsprogramm für VHDL [14].

	Zeit (normiert)	
Technik	**K1**	**K2**
Simulations-Kommandos	1.00	1.00
Mutanten mit Kommandos	1.01	1.23
Mutanten	1.05	1.96
Saboteure mit Kommandos	1.28	1.81
Saboteure	1.30	1.84
FIT (VHDL Fault Injection Tool)	3.71	5.21

Tabelle 2: Simulationsvergleich verschiedener Fehlerinjektionen

3. Switch-Level-Modelle in SystemC

Im Abschnitt zuvor wurde einige Techniken zur Simulation von Fehlern vorgestellt. Neben der Fehlerinjektionstechnik ist noch ein weiterer Punkt interessant. Die Modellierung der Hardware bezüglich der Entwurfsebene ist es genauso wert, näher untersucht zu werden. Für verhaltensorientierte Simulationen ist das Transaction Level Modeling (TLM) sehr populär [16] und Modelle des RT-, und Gate-Levels sind z.B. in [17] gut beschrieben. Um jedoch SystemC-Modelle auf dem switch-level zu modellieren sind einige Erweiterungen erforderlich. In eigenen, früheren Arbeiten wurden Komponenten erstellt, die mittels ergänzender Funktionen z.T. ein switch-level-Verhalten verhaltensorientiert nachbildeten [19]. Eine universellere und allgemein besser zu verwendende Methode wird im Folgenden präsentiert.

Für eine realistischeres HW-Verhalten wurde der Datentyp sc_logic erweitert. Die Schritte hierzu waren dabei ähnlich der, wie sie unter 2.3 beschrieben sind. Die neuen Logiktypen wurden hierbei IEEE 1164 [18] konform erstellt. Der neue Werteumfang ist somit:

$$sc\_logic \in \{0, 1, Z, X, L, H, U, W, D\}.$$

Die Signalstärken wurden entsprechend der IEEE 1164 in der *sc_signal-resolved*-Klasse implementiert.

Auf dem switch-level werden Elemente als Transistoren mit Schaltereigenschaften betrachtet. Des Weiteren sind switch-level-Modelle fähig, einen bidirektionalen Informationstransfer widerzuspiegeln. Die zwei Transistorarten *nmos* und *pmos* waren für unsere Modellbildung ausreichend. Der *nmos*-Transistor ist bei einem High-Pegel an seinem Gate-Anschluss leitend und der *pmos*-Transistor bei einem Low-Pegel.

Die Abbildung 5 zeigt einen Teil der Umsetzung für einen nmos-Transistor. Die Verhaltensmodellierung erfolgte in Anlehnung an [15].

```
   const char nmosTab[4][4] =  {
2  //   0    1    Z    X
     { 'Z', '0', 'L', 'L' }, //0
4    { 'Z', '1', 'H', 'H' }, //1
     { 'Z', 'Z', 'Z', 'Z' }, //Z
6    { 'Z', 'X', 'X', 'X' }, //X
   };
8
   void nmos::doit()
10 {
    a0_l = a0.read();
12  a0_t = (int) a0_l.value();
    ctrl_l = ctrl.read();
14  ctrl_t = (int)ctrl_l.value();
    z0.write((sc_logic)
16          nmosTab[a0_t][ctrl_t]);
    . . .
```

Abbildung 5: nmos-Transistor

Die Funktionalität wird durch die Tabelle *nmosTab* realisiert. Die Tabellen-Variante bietet einen schnelleren Zugriff im Vergleich zu einer *switch-case*-Abfrage. Die Methode *doit* gewährleistet den korrekten Zugriff und die entsprechende Typanpassung. Für einen pmos-Transistor sieht die Implementierung ähnlich aus.

Mit Hilfe der Schalter und einer Hierarchiebildung lassen sich die meisten gate-level-Modelle abbilden. Ein wesentlicher Vorteil ist die genauere Wiedergabe des realen Verhaltens. Besonders deutlich wird dies bei der Simulation von Fehlern. Von uns wurde es für Transmission Gates (einem bidirektionalem Schalter), Standard-, Pass-Transistor- und reversible Logik modelliert.

Die Tabelle 3 zeigt die Simulationsergebnisse einiger Schaltungen. Hierbei bestehen TGB1, TGB2 und MUX16 überwiegend aus Transmission Gates und K1 aus Standard-Logik.

Design level	**Simulationszeiten (normiert)**			
	TGB1	***TGB2***	***MUX16***	***K1***
Gate level	1	1	1	1
Switch level	1,07	1,54	1,05	1,82

Tabelle 3: Simulationsvergleich zwischen gate- und switch-level

Die Simulationszeiten für die switch-level-Implementierungen fallen erwartungsgemäß höher aus, liegen aber noch im akzeptablen Bereich. Die dargestellten Ergebnisse gelten für die funktionale Simulation. Für eine Simulation mit Fehlerinjektion erhöht sich der Zeitaufwand nahezu proportional zum Wachstum der Schaltungsknoten. Für die Beispiele in der Tabelle 3 ist die Simulation mit Fehlern immer

noch zu vertreten, da bei den verwendeten Modellen der Zuwachs an Schaltungsknoten kleiner 2 ist.

3. Zusammenfassung

Es wurde gezeigt, dass einige Fehlerinjektionstechniken, wie sie in anderen HW-Beschreibungssprachen schon erfolgreich angewandt wurden, auch in SystemC möglich sind. Es wurde auch gezeigt, dass Fehlerinjektionen für die es kein SystemC-Äquivalent gibt (simulator commands), durch Erweiterungen der SystemC-Bibliothek u.ä. nachbildbar sind. Die neuen Implementierungen überzeugen durch eine höhere Simulationsgeschwindigkeit. Die vorgestellten Techniken der Fehlerinjektionen entstanden zunächst im Hinblick auf HW-genaue Modelle, sie lassen sich aber prinzipiell auch auf den höheren Ebenen des Systementwurfs verwenden.
Des Weiteren wurden SystemC-Modelle erstellt, die auf dem switch-level agieren. Die Simulationszeiten einiger Schaltungen zeigten zwar einen Mehraufwand, dieser bewegte sich aber immer noch in einem akzeptablen Bereich.

5. Referenzen

[1] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero: RT-level Fault Simulation Techniques based on Simulation Command Scripts, DCIS2000: XV Conference on Design of Circuits and Integrated Systems, Le Corum, Montpellier, November 21-24, 2000, pp. 825-830

[2] E. Jenn, J. Arlat, M. Rimen. J. Ohlsson, J. Karlsson: Fault Injection into VHDL Models: The MEFISTO Tool, Proc. 24th International Symposium Fault-Tolerant Computimg, IEEE, 1994 pp. 66-75.

[3] J. Gracia, J.C. Baraza, D. Gil, P.J. Gil: Comparison and Application of Different VHDL-Based Fault Injection Techniques, IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'01), October 2001, pp. 0233.

[4] M. Rimén, J. Ohlsson, J. Karlsson, et al.: Design Guidelines of a VHDL-based Simulation Tool for the Validation of Fault Tolerance 1997

[5] Alessandro Fin, Franco Fummi, Graziano Pravadelli: AMLETO: A Multi-language Environment for Functional Test Generation, Proceedings International Test Conference 2001, pp. 821

[6] F. Bruschi, M. Chiamenti, F. Ferrandi, D. Sciuto: Error Simulation Based on the SystemC Design Description Language, Design, Automation and Test in Europe Conference and Exhibition (DATE'02) 2002, pp. 1135

[7] K. Rothbart, U. Neffe, Ch. Steger, R. Weiss, E. Rieger, A. Muehlberger: High Level Fault Injection for Attack Simulation in Smart Cards, Found in: 13th Asian Test Symposium (ATS'04), November 2004, pp. 118-121

[8] SystemC: www.systemc.org, 1.03.2006.

[9] R. Le Moigne, O. Pasquier, J-P. Calvez: A Generic RTOS Model for Real-time Systems Simulation with SystemC, Forum DATE'04, 2004, pp. 30082

[10] Mentor Graphics Inc.: Modelsim, www.modelsim.com, 05.02.2007

[11] Aldec Inc.: Active-HDL und Riveria, www.aldec.com, 1.11.2005

[12] Dirk Louis: C / C++ Professionell Kompendium, Markt und Technik München 2004, ISBN 3-827-26812-5

[13] Perl: www.perl.org, 02.04.1007

[14] Galke, C.; Misera, S.; Fröschke, H.; Vierhaus, H. T.: "Eine Simulationsumgebung zur Validierung des Fehlerverhaltens für Prozessor-basierte Systeme " - Poster, Proc. 17. ITG-GI-GMM-Workshop "Test und Zuverlässigkeit von Schaltungen und Systemen", Innsbruck, 2005

[15] http://standards.ieee.org/reading/ieee/std_public/description/dasc/1364-1995_desc.html

[16] F. Ghenassia, Ed., Transaction-Level Modeling with SystemC. TLM Concepts and Applications for Embedded Systems. Springer, June 2005, ISBN 0-387-26232-6

[17] J. Bhasker: A SystemC Primer, Star Galaxy Publishing Allentown 2002, ISBN 0-9650391-8-8

[18] IEEE Package 1164: http://ieeexplore.ieee.org/xpl/standards.jsp, 29.01.2007.

[19] Misera, S.; Sieber, A.: "Hardware-nahe Fehlersimulation mit effektiven SystemC-Modellen", 10. GI/ITG/GMM Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen", Erlangen, 2007

[20] Black, Donovan: SystemC: From The Ground Up, Kluwer Academic Publishers, Boston 2004.