Ein Template-basierter Ansatz zur automatisierten Generierung von SystemC-Modellen aus IP-XACT-Beschreibungen

Stefan Müller, Yumin Zhou, Axel Braun,
Joachim Gerlach, Wolfgang Rosenstiel

Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Arbeitsbereich Technische Informatik
{muellers|abraun|gerlach|rosenstiel}@informatik.uni-tuebingen.de

ZUSAMMENFASSUNG

Beim Entwurf komplexer Systems-on-Chip (SoC) kommt der effektiven Wiederverwendung von bereits vorhandenen Entwurfskomponenten (Intellectual Property, IP) eine zentrale Bedeutung zu. Systementwürfe basieren typischerweise zu einem Großteil oder auch vollständig auf bereits vorhandenen Systembausteinen (z.B. Prozessoren und Speichermodule externer Hersteller, standardisierte Schnittstellen, etc.). Entsprechend stellt eine effektive Wiederverwendung von IP-Modulen im Entwurfsablauf, insbesondere bereits in einer frühen Phase des Systementwurfs, eine notwendige Voraussetzung dar. Hierfür wurde mit dem Beschreibungsstandard IP-XACT eine Methodik geschaffen, um eine handhabbare und Ebenen-übergreifende Beschreibung der Schnittstelleneigenschaften von IP-Modulen zu erreichen. Die Sicherstellung der Modellkonsistenz, etwa die konsistente Handhabung von Schnittstellenregistern beim Zusammenspiel von Hardware- und Softwarekomponenten, sowie deren Konsistenthaltung, etwa bei Änderungen der Spezifikation oder bei der Durchführung von Entwurfsschritten, ist eine anspruchsvolle Herausforderung. Hierzu wurde ein Ansatz entwickelt, der durch die Verwendung von flexibel anpassbaren Templates prinzipiell unabhängig von der verwendeten Spezifikationssprache ist. Die Templates lassen sich sowohl für eine hardware- als auch softwareorientierte Anwendung konfigurieren. Damit können bei busbasierten Systemmodellen sowohl die Hardwarestrukturen der Registerfelder erzeugt werden, als auch die Header-Dateien für die Entwicklung von Software-Treibern zur Verfügung gestellt werden. Die automatische Generierung dieser Informationen aus einer einzigen IP-XACT-Basisspezifikation sichert die Konsistenz zwischen Hardware- und Softwarespezifikationen zu.

1. EINFÜHRUNG

Beim Entwurf von SoC's spielt die Verwendung bzw. Wiederverwendung von IP-Komponenten eine immer stärkere Rolle. So bestehen neue Designs meist zu einem großen Teil oder sogar komplett aus eigenen bzw. zugekauften IP-Komponenten, wie zum Beispiel Prozessoren, Speicher, Schnittstellen, etc.[1] Um die Schnittstellen der einzelnen IP-Komponenten und des daraus aufgebauten Gesamtsystems zu beschreiben wurde der IP-XACT Standard [2] geschaffen. In IP-XACT-Dateien werden die Interfaces der einzelnen IP-

Blöcke, jedoch aber nicht deren Funktion, beschreiben. Dieser Standard soll die Integration und den Austausch von IP-Komponenten vereinfachen. Es lassen damit aber auch grundlegende Testumgebungen generieren [3]. Die Beschreibung der IP-Komponenten ist dabei sehr auf den Register-Transfer-Level (RTL) fokussiert. Jedoch lassen sich daraus auch Informationen für abstrakte Systemmodelle, wie zum Beispiel SystemC TLM-Modelle und die Software- bzw. Treiberentwicklung, wie zum Beispiel Header-Dateien, beziehen. Wenn nun alle in der Entwicklung verwendeten Modelle und Spezifikationen als Basis die Informationen der IP-XACT Beschreibung verwenden, kann dies in vielen Punkten hilfreich sein. So können beispielsweise bei einem Respin, welche in den frühen Entwicklungsphasen eines SoC's durchaus mehrfach auftreten können, alle Modelle und Codes einfach auf die neue Beschreibung angepasst werden. Somit wird die Konsistenz über alle Entwicklungsphasen und Modelle hinweg sichergestellt.

Das Paper ist wie folgt gegliedert: Abschnitt 2 befasst sich mit den Template- und Pattern-Strukturen, die für die Generierung von Code- und Konfigurationsdateien notwendig sind. In Abschnitt 3 wird das hieraus entwickelte Generierungswerkzeug vorgestellt. Abschnitt 4 zeigt Anwendungsfälle für diesen Ansatz auf. In Abschnitt 5 erfolgt eine Zusammenfassung, sowie ein Ausblick.

2. TEMPLATES UND PATTERN

Im Entwurf von SoCs kommen verschiedene Spezifikationssprachen, Entwicklungs- und Testumgebungen für die verschiedenen Entwurfsphasen und -domänen zum Einsatz. So werden in den frühen Phasen verstärkt SystemC-Modelle verwendet, die zum Teil an Instruction-Set-Simulatoren (ISS) oder andere Simulationsumgebungen angebunden sind. In den späteren Phasen werden diese dann durch VHDL- oder Verilog-Modelle konkretisiert. Wenn sich nun bei einem Respin zum Beispiel die Struktur von verschiedenen Registern einer Komponente ändert, hat das Auswirkungen auf alle erstellten Modelle und die Konfigurationen der angebunden Simulationsumgebung. So müssen in allen Modellen die Registeradressen angepasst werden. Wenn ein ISS oder eine Co-Simulationsumgebung verwendet wird, müssen hier die Registeradressen in der ISS- bzw. Co-Simulationskonfiguration abgeändert werden. Das gleiche gilt auch für Testumgebungen. Diese Änderungen sind manuell nur sehr zeitaufwändig und fehleranfällig durchzuführen.

Mit der Einführung von IP-XACT wurde ein Format standardisiert, welches die Schnittstellen eines Systems beschreibt. Hierzu gehören zum Beispiel Registerfelder, die z.B. alle auf einem Bussystem adressierbaren Register enthalten. Durch die Verwendung von Templates ist es möglich aus diesen IP-XACT-Informationen die unterschiedlichen Spezifikationen und Konfigurationen zu erzeugen, die in einem Projekt verwendet werden. Ein Problem bei der Verwendung von generierten Spezifikationen ist die erneute Generierung. Werden manuelle Änderungen und Anpassungen vorgenommen, werden diese bei der erneuten Generierung verloren gehen.

Um nach einer Änderung der IP-XACT-Daten nicht diese Erweiterungen zu verlieren müssen die Codes, wie in Abbildung 1 gezeigt, mehrschichtig aufgebaut sein. Die erste Ebene enthält dabei alles, was aus den IP-XACT-Daten generiert werden kann, zum Beispiel Register,

Busschnittstellen und Trigger, die beim Registerzugriff ausgelöst werden. In der zweiten Ebene befinden sich jene Teile, die nicht aus den IP-XACT-Daten generiert werden können: Dazu zählen die Logik, Verarbeitungsalgorithmen und Trigger-Methoden, die bei den Zugriffen auf ein Register ausgeführt werden.

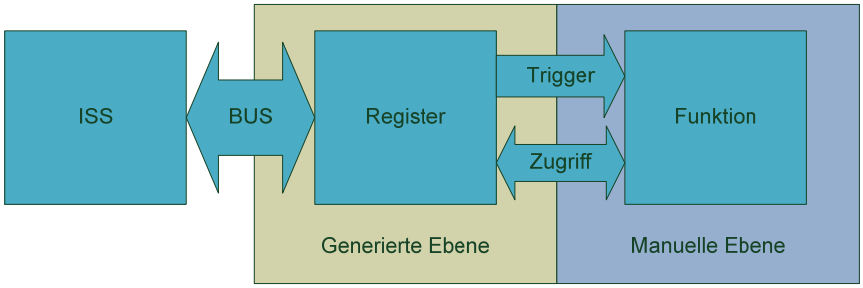

Abbildung 1: Zwei-Ebenen-Modell für Code

Durch diese Trennung der aus der IP-XACT-Beschreibung bekannten Interfaces wie Register und Busschnittstellen in eine generierte Ebene und den nicht bekannten Funktionen in eine manuelle Ebene, kann die generierte Ebene unabhängig von den zugehörigen Komponenten wie den Modellen der restlichen Busses oder Funktionen des Moduls neu generiert werden. Änderungen der Schnittstellen – wie zum Beispiel neue Registerfelder – haben neue Trigger zur Folge, die in der manuellen Ebene implementiert sein müssen. Ohne sie schlägt ein erneutes Erstellen des Gesamtmodells fehl und erzeugt somit Inkonsistenzen und führt somit zu weitreichenden Änderungen – eventuell im gesamten System.

Diese Methode zur Trennung der bekannten/generierbaren und nicht bekannten/generierbaren Bestandteile der Modellbeschreibung funktioniert gut für Szenarien in denen Abhängigkeiten über mehrere Dateien hergestellt werden können, wie Programmiersprachen z.B. C++ über Vererbung oder Includes. Bei Konfigurationsdateien ist die Möglichkeit die Konfiguration über mehrere Dateien zu verteilen und somit die oben gezeigte Trennung zu erreichen, oft nicht gegeben und macht die Generierung dieser Dateien schwieriger. Um diese Hürde zu nehmen reichen Templates allein nicht aus. Hier kann die Generierbarkeit aber über die Erweiterrung um eine Abstraktionsebene erreicht werden. Diese Abstraktion erfolgt durch die Verwendung einer Script- oder Makrosprache wie Python, Perl oder m4. Diese Scripte können so generiert werden, dass sie die ursprüngliche Konfiguration einlesen und an den entsprechenden Stellen modifizieren. Wie in Abbildung 2 gezeigt, wird hierzu ein Template für ein Script entworfen, welches die generierten Informationen in der Konfigurationsdatei kennt und bei seiner Ausführung ersetzt.

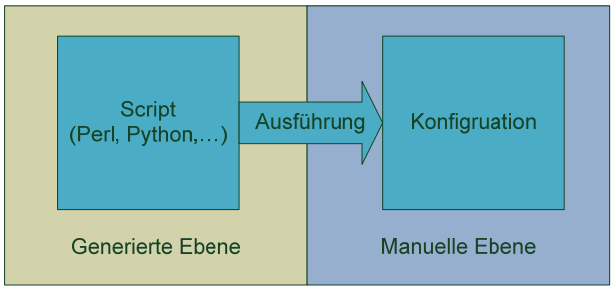

Abbildung 2: Zwei-Ebenen Modell für Konfigurationsdateien

Diese Methode wird am Beispiel eines SystemC-Modells eines einfachen Moduls mit Registern im Abschnitt 4 verdeutlicht.

3. WERKZEUG

3.1. Aufbau

IP-XACT-Beschreibungen von ganzen SoCs können mehrere tausend Zeilen enthalten und damit unhandlich sein. Um den Umgang mit solchen Dateien zu erleichtern wurde ein Werkzeug entwickelt, welches den Umgang mit IP-XACT in der Eclipse Entwicklungsumgebung erleichtert und die Generierung von Code- und Konfigurations-dateien erlaubt. Das Werkzeug besteht aus den Eclipse Plugins für einen Template-Editor und einen Wizzard. Der Wizzard erlaubt es in Eclipse-Projekten, aus einer IP-XACT-Beschreibung und den vorhandenen Templates, die entsprechenden SystemC oder Konfigurationsdateien zu generieren.

Für die Erstellung des Tools wurde das Eclipse Modelling Framework (EMF) [4][5] verwendet. EMF ist ein Framework, welches nach dem Model-Driven Architecture (MDA) Paradigma arbeitet [6]. Hierbei wird die Software durch Datenmodelle und Richtlinien definiert und anschließend in Code umgewandelt. Als Quelle können XML-Schema-Definition (XSD), Unified Modelling Language (UML) Beschreibungen oder annotierter Java-Code verwendet werden. EMF generiert aus diesen Quellen ein Ecore-Modell, welches eine API für den Zugriff auf die Datenstruktur liefert. Diese API wird in unserem Explorer, welcher in 3.2 genauer beschrieben wird, und für die Generierung der Templates verwendet. Accellera [7] stellt für IP-XACT mehrere XSD-Dateien bereit. Mit Hilfe des EMF wurde aus diesen Schema-Definitionsdateien ein Ecore-Metamodell generiert. Dieses Ecore-Modell dient als Basis für das Tool.

Das Java Emitter Template (JET) ist eine Modell-2-Text Engine, welche das Erzeugen von Textausgaben – basierend auf einem EMF-Modell – erlaubt. Dabei spielt es keine Rolle um welche Art von Text, z.B. Java, XML, SystemC, VHDL, etc., es sich handelt. Um die Performanz der Verarbeitung von großen Modellen zu erhöhen werden die Template-Dateien in Java-Implementierungen übersetzt. Die Templates verwenden drei Arten von Ausdrücken – Directives, Expressions und Scriptlets. Die Directives enthalten die Einstellungen für das JET-Template, wie beispielsweise den Import von Java-Klassen und -Packages. Expressions erlauben das Einfügen von Strings in den Ausgabetext, wie zum Beispiel die Werte aus den IP-XACT-Daten. Scriptlets können jegliche Art von Java-Code enthalten, der Ergebnisse für die Textausgabe erzeugt.

3.2. FUNKTION

Der Template-Editor erlaubt es neue Templates zu erstellen oder existierende Templates anzupassen. Um die Erstellung der Templates zu erleichtern kann eine IP-XACT-Beschreibung in ein Explorer-Fenster (Abbildung 3) geladen werden. Dies erleichtert zum einen das Lesen der IP-XACT-Beschreibung durch die Darstellung der Informationen in einer Baustruktur, welche auch die Navigation erleichtert. Zum anderen, hilft der Explorer aber auch bei der Suche nach bestimmten Bereich oder Einträgen in der Beschreibung. Durch seine

Filtermöglichkeit im oberen Bereich der Baumansicht kann ein Suchtext eingegeben werden, nach dem der Baum durchsucht wird und alle anderen Teile ausgeblendet werden.

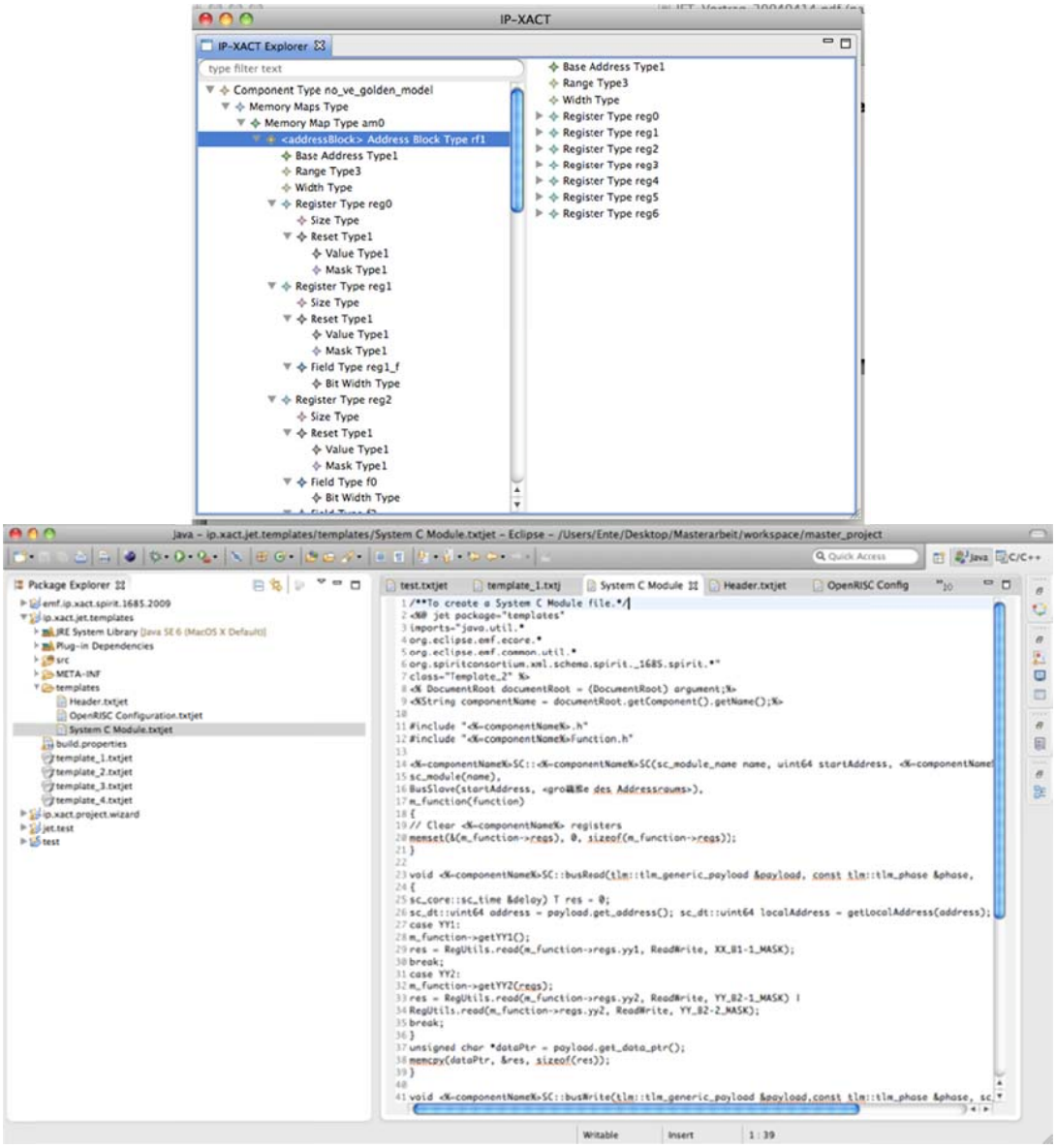

Abbildung 3: Template-Editor und IP-XACT Explorer

Mit dem Wizzard (Abbildung 4) kann in einem Eclipse-Projekt eine neue. aus einer IP-XACT-Beschreibung generierte, Datei hinzugefügt werden. Dafür muss zunächst die IP-XACT-Beschreibung ausgewählt werden. Im nächsten Schritt werden alle erstellten Templates angezeigt, die für diese IP-XACT-Beschreibung zur Generierung geeignet sind. Templates, die auf Einträge der Beschreibung verweisen, die in der aktuell ausgewählten Beschreibung nicht vorhanden sind, werden ausgeblendet. Wenn für die Generierung im Template Optionen eingefügt wurden, wie zum Beispiel, ob Monitoring hinzugefügt oder welcher Bustyp verwendet werden soll, können diese noch ausgewählt werden. Im nächsten Schritt wird abschließend der Dateinamen der zu generierenden Datei festgelegt und im Anschluss die Spezifikationsdateien nach den ausgewählten Optionen generiert.

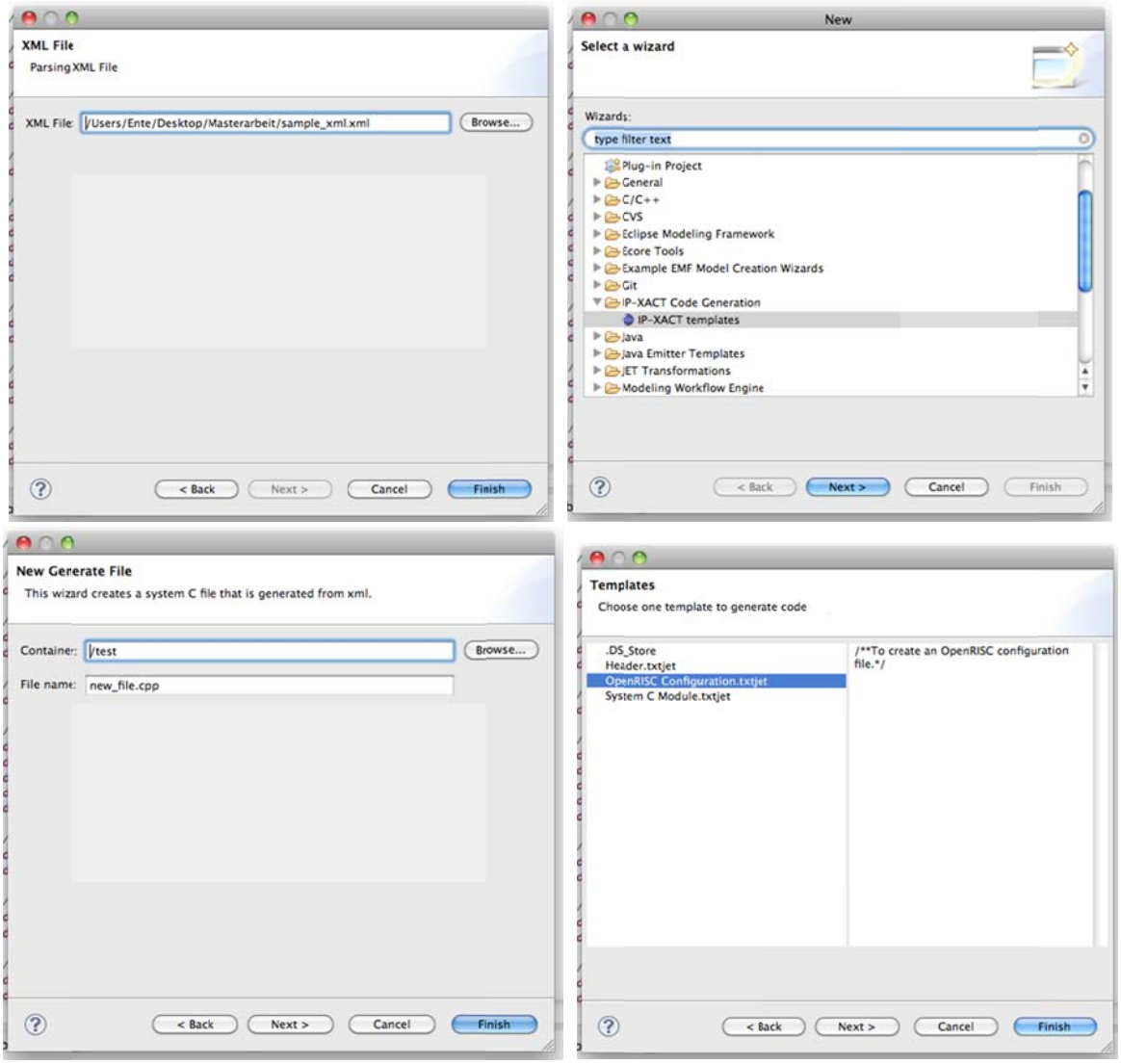

Abbildung 4: Wizzard zur Dateigenerierung

4. ANWENDUNGSFÄLLE

4.1. SYSTEMC

Die Trennung des generierten Spezifikationscodes vom manuell erzeugten Funktionscode wird in SystemC mittels zweier Klassen erreicht – wie in Abbildung 1 gezeigt wird. Der generierte Code befindet sich dabei vollständig in der Klasse `no_ve_golden_modelSC`. Diese definiert auch den gezeigten Datentyp `registers` in Form einer `struct`. `uint8_t` steht hier exemplarisch für die Datenbreite des jeweiligen generierten Registerfelds. Die Kommunikation zwischen dem Bus und dem generierten Modul `no_ve_golden_modelSC` wird in unserem Fall über die Klasse BusSlave abgehandelt. Diese stellt die virtuellen Methoden `busRead` und `busWrite` zur Verfügung, welche in `no_ve_golden_modelSC` mit dem generierten Code gefüllt werden und die Trigger in `no_ve_golden_modelFunction` aufrufen, um den manuell erzeugten Code auszuführen.

Abbildung 5: Klassendiagramm SystemC Modul Generierung

Um die Trigger (`getreg0`, `setreg0`,…) vor dem ungewollten Zugriff von anderen Klassen zu schützen sind diese `protected`. Die generierte Klasse `no_ve_golden_modelSC` erhält durch eine Friend-Beziehung mit `no_ve_golden_modelFunction` Zugriff auf diese Trigger. Wie im Codebeispiel Listing 1 gezeigt wird kann durch eine Helperklasse wie `RegUtils` der Code sehr übersichtlich und einfach generiert werden. Die Helperklasse kümmert sich dabei die Lese- und Schreibzugriffe auf die richtigen Registerfelder unter Beachtung der Zugriffsrechte. Dafür bekommt sie das jeweilige Register, die Zugriffsoperation und eine Bitmaske, die den Bereich des Registers angibt wenn dieses in mehrere Felder unterteilt ist. Bei Schreibzugriffen bekommt sie noch den zu schreibenden Wert übergeben.

```
void no_ve_golden_modelSC::busRead(tlm::tlm_generic_payload &payload,
   const tlm::tlm_phase &phase,
   sc_core::sc_time &delay)
{
   uint8_t res = 0;

   // extract data from payload
   switch (localAddress)
   {
      // more registers
      case reg4:
         m_function->getreg4();
         res= RegUtils.read(m_function->regs.reg4, ReadWrite,
                 no_ve_golden_model_f0_MASK)
            | RegUtils.read(m_function->regs.reg4, ReadWrite,
                 no_ve_golden_model_f1_MASK)
            | RegUtils.read(m_function->regs.reg4, WriteOnly,
                 no_ve_golden_model_dirs_MASK)
```

```
            | RegUtils.read(m_function->regs.reg4, ReadOnly,
                no_ve_golden_model_more_dirs_MASK)
            | RegUtils.read(m_function->regs.reg4, ReadWrite,
                no_ve_golden_model_rsv_MASK);
         break;

      // more registers
      }

      // put reg value into payload
}

void SPISlaveSC::busWrite(tlm::tlm_generic_payload &payload,
   const tlm::tlm_phase &phase,
   sc_core::sc_time &delay)
{
   // extract data from payload

   switch (localAddress)
   {
      // more registers

      case reg4:
         RegUtils.write(m_function->regs.reg4, ReadWrite, no_ve_golden_model_f0_MASK, wdata);
         RegUtils.write(m_function->regs.reg4, ReadWrite, no_ve_golden_model_f1_MASK, wdata);
         RegUtils.write(m_function->regs.reg4, WriteOnly, no_ve_golden_model_dirs_MASK,
            wdata);
         RegUtils.write(m_function->regs.reg4, ReadOnly, no_ve_golden_model_more_dirs_MASK,
            wdata);
         RegUtils.write(m_function->regs.reg4, ReadWrite, no_ve_golden_model_rsv_MASK, wdata);
         m_function->setCONFIG();
         break;

      // more registers
   }
}
```

Listing 1: Generierter SystemC Code

4.2. ISS

Beispielhaft für die Einbindung eines ISS verwenden wir den OpenRISC ISS, welcher frei verfügbar ist [8]. An diesen wird das generierte Modul über eine TLM Schnittstelle mit dem im OpenRISC verwendeten Whishbone-Bus verbunden. Durch das Einfügen von zwei Kommentaren können die Blöcke für die Modul-beschreibungen zwischen den beiden Kommentaren immer wieder gefunden werden. Ein Eintrag für unser Modul sieht in der OpenRISC Konfiguration wie in Listing 2 aus. Alle Module die nicht vom ISS stammen werden als generic eingefügt. Diese benötigen mindestens die Startadresse des Moduls (baseaddr), die Größe des Moduls (size), mit welcher Busbreite auf das Modul zugegriffen werden kann ({byte,hw,word}_enable) und einen Name für Debugging Zwecke. All diese Informationen sind bereits in der IP-XACT-Beschreibung vorhanden und können somit vollständig generiert werden.

```
/* begin generated modules */
section generic
  enabled         =          1
  baseaddr        =     0x1000
  size            =      0x100
  byte_enabled    =          1
  hw_enabled      =          0
  word_enabled    =          0
  name            = "no_ve_golden_model"
end
/* end generated modules */
```

Listing 2: OpenRisc Konfiguration

4.3. TREIBER

Um das Modul in einem OpenRISC-Programm verwenden zu können benötigt man zumindest die Adressen der Register. Je nach Compiler können die Adressen mit einem Schlüsselwort mit einem Namen versehen werden. Zu diesem Zweck kann eine Headerdatei generiert werden die das Mapping zwischen Registernamen und Adressen herstellt. Eine Headerdatei für das im Beispiel verwendete SystemC-Modul sieht wie in Listing 3 aus. Es erlaubt den Zugriff auf die einzelnen Register über eine Strukur `no_ve_golden_model_t`. Hier könnten auch noch die Bit-Masken für die Zugriffe auf die Registerfelder generiert werden.

```
struct no_ve_golden_model_t{
	volatile uint8_t reg0;
	volatile uint8_t reg1;
	volatile uint8_t reg2;
	volatile uint8_t reg3;
	volatile uint8_t reg4;
	volatile uint8_t reg5;
	volatile uint8_t reg6;
};

volatile struct no_ve_goleden_model_t *no_ve_golden_model;

void init_no_ve_golden_model()
{
	no_ve_golden_model = (struct no_ve_goleden_model_t *)0x1000;
}
```

Listing 3: Headerdatei für einen OpenRisc Treiber

5. ERGEBNIS UND AUSBLICK

Das hier gezeigte Werkzeug kann für verschiedene Domänen und Phasen in SoC Entwurfsprojekten verwendet werden. Durch seine flexible Model-2-Text Engine lassen sich auch komplexere Ausgabedateien erzeugen. Das vorgestellte Zwei-Ebenen-Konzept ermöglicht eine saubere Trennung der, aus einer IP-XACT-Beschreibung stammenden, generierten Anteile und der manuell durch den Benutzer erstellten Anteile. Beispielhaft wurde gezeigt, wie man die Konsistenz bei Modellen gewährleistet die mit einem ISS, in diesem Fall dem des OpenRISC-Cores, verbunden sind. Des Weiteren wurde gezeigt wie man die IP-XACT-Interfaceinformationen auch auf der Seite der Softwareentwicklung zur Generierung von Headerdateien für die Treiberentwicklung nutzen kann. In zukünftigen Entwicklungen soll die die Erweiterung dieser Methodik, um die automatische Generierung von Monitoring- und Injektionspunkten in den Modellen untersucht werden. Ziel ist es neben dem Release-Modell, welches hier gezeigt wurde, auch ein Debug-Modell zu haben, welches die gleichen Eigenschaften aufweist.

6. DANKSAGUNG

Diese Arbeit wurde mit Mitteln des Ministeriums für Wissenschaft, Forschung und Kunst Baden-Württemberg im Rahmen des kooperativen Promotionskollegs EAES gefördert.

LITERATUR

[1] H. Chang, L. R. Cooke, M. Hunt, G. Martin, A. McNelly, and L. Todd, *Surviving the SOC Revolution - A Guide to Platform-Based Design*. Springer, 1999, p. 256.

[2] "Design Automation Standards Committee," *IEEE SA - 1685-2009 - IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows*. 2009.

[3] T. Xie, W. Mueller, and F. Letombe, "IP-XACT based system level mutation testing," in *2011 IEEE International High Level Design Validation and Test Workshop*, 2011, pp. 65–71.

[4] The Eclipse Foundation, "Eclipse Modeling - EMF - Home." [Online]. Available: http://www.eclipse.org/modeling/emf/. [Accessed: 13-Nov-2012].

[5] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley Professional, 2008, p. 704.

[6] S. J. Mellor, K. Scott, A. Uhl, and D. Weise, "Model-Driven Architecture," in *Advances in Object-Oriented Information Systems*, vol. 2426, J.-M. Bruel and Z. Bellahsene, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 290–297.

[7] "IP-XACT - Accellera Systems Initiative." [Online]. Available: http://www.accellera.org/activities/committees/ip-xact. [Accessed: 14-Nov-2012].

[8] "OR1200 OpenRISC Processor - OR1K□:: OpenCores." [Online]. Available: http://opencores.org/or1k/OR1200_OpenRISC_Processor. [Accessed: 14-Nov-2012].