COMPUTING
SCIENCE

Formal Software and Hardware Development: A Case
Study in the use of CSDM, SPECTRUM and HOLCF

L.J. Steggles and M. Wirsing

TECHNICAL REPORT SERIES

No. 551 July, 1996
Formal Software and Hardware Development: A Case Study in the use of CSDM, SPECTRUM and HOLCF

L.J. Steggles and M. Wirsing

Abstract

We present a simple methodology for the formal verification of software and hardware systems using algebraic specification methods. This methodology is based on the notion of functional refinement and we define the conditions necessary for a design specification to be a correct functional refinement of an abstract requirement specification. In particular we consider facilitating the reuse of design specifications and the role of the environment information contained within the requirement specification during the verification process. We demonstrate our verification methodology by considering the formal verification of two systolic algorithms for computing the convolution function. This case study is carried out within the CSDM software development environment using the specification language SPECTRUM and the theorem prover Isabelle with the object logic HOLCF.
Bibliographical details

STEGGLE, Lance Jason

Formal Software and Hardware Development: A Case Study in the use of CSDM, SPECTRUM and HOLCF
[By] L.J. Steggle and M. Wirsing


(University of Newcastle upon Tyne, Computing Science, Technical Report Series, no. 551)

Addt entries

UNIVERSITY OF NEWCASTLE UPON TYNE.
WIRILING, Martin.

Abstract

We present a simple methodology for the formal verification of software and hardware systems using algebraic specification methods. This methodology is based on the notion of functional refinement and we define the conditions necessary for a design specification to be a correct functional refinement of an abstract requirement specification. In particular we consider facilitating the reuse of design specifications and the role of the environment information contained within the requirement specification during the verification process. We demonstrate our verification methodology by considering the formal verification of two systolic algorithms for computing the convolution function. This case study is carried out within the CSDM software development environment using the specification language SPECTRUM and the theorem prover Isabelle with the object logic HOLCF.

About the author

Dr. L. J. Steggle studied for the degree of Ph.D. in theoretical computer science at the University of Wales, Swansea. He has been a full-time demonstrator in the Department of Computing Science at the University of Newcastle upon Tyne since January 1995.

Prof. M. Wirsing is the head of the Institute fuer Informatik at the Ludwig-Maximilians-Universitaet in Munich.

Suggested keywords

ALGEBRAIC SPECIFICATION
DESIGN METHODOLOGY
FORMAL METHODS
SOFTWARE TOOLS

Suggested classmarks (primary classmark underlined)

Dewey (18th): 512 511.3
U.D.C. 512.57 510.64
Formal Software and Hardware Development:
A Case Study in the use of CSDM,
SPECTRUM and HOLCF

L. J. Stiegles1 and M. Wirsing2.
1Department of Computer Science, University of Newcastle.
email: l.j.stiegles@newcastle.ac.uk
2Institute of Computer Science, Ludwig-Maximilians-University of Munich.
email: wirsing@informatik.uni-muenchen.de

Abstract
We present a simple methodology for the formal verification of software and hardware systems using algebraic specification methods. This methodology is based on the notion of functional refinement and we define the conditions necessary for a design specification to be a correct functional refinement of an abstract requirement specification. In particular we consider facilitating the reuse of design specifications and the role of the environment information contained within the requirement specification during the verification process. We demonstrate our verification methodology by considering the formal verification of two systolic algorithms for computing the convolution function. This case study is carried out within the CSDM software development environment using the specification language SPECTRUM and the theorem prover Isabelle with the object logic HOLCF.

1 Introduction.
It has long been recognized that the use of formal mathematical techniques in the design of complex software and hardware systems can help ensure the correctness and reliability of the resulting system. With the dramatic growth in the use of increasingly complex computer systems, often in safety critical applications where failure can have lethal consequences, there has been increasing interest in so called formal methods. A variety of different formal methods can be found in the literature (for an overview see for example Cohen et al. [1986]). Algebraic specification methods are a theoretically well-founded formal method for computer system design, supporting formal specification, verification, modularization, data abstraction, program transformation and refinement. The literature on these methods is extensive: recent surveys are Ehrig [1994] and Wirsing [1990], while an introduction to the methods can be found in Goguen et al. [1978] and Ehrig and Mahr [1985, 1990].

However, in order for algebraic specification methods to be accepted as a practical formal method further research is needed into the methodology of using these techniques in practice and the integration of software tools into the formal design process. In this paper we propose a simple methodology for the formal verification of computing systems using (higher-order) algebraic specification methods. This verification methodology is based on the notion of functional refinement (see for example Broy and Wirsing [1995]) and involves the construction of: a statement specification which simple names the components to be implemented; a requirement specification which abstractly specifies the components given in the statement specification and contains a distinguished environment part referred to as the frame specification; and a design specification
which specifies a proposed implementation of the components given in the statement specification. The environment information contained within the requirement specification can be seen as providing a context in which the implementation, as specified by the design specification, will be used. Thus before verifying the correctness of the design specification we first instantiate it with this environment information. This approach has the benefit of allowing design specifications to abstract away from specific environment details and thus facilitates the reuse of design specifications. However, it is important to note that a design specification is only verified correct with respect to the particular environment given by the requirement specification. In order to verify that the design specification is a correct functional refinement of the requirement specification we must show that the following three conditions are satisfied:

(i) the consistency condition which states that the requirement and design specifications must be consistent specifications;
(ii) the preservation condition which ensures that the environment information contained within the requirement specification is preserved when it is combined with the design specification; and
(iii) the refinement condition which states that all the axioms of the requirement specification must be valid in the semantics (the class of models) of the design specification augmented with the environment information provided by the requirement specification.

We demonstrate our verification methodology by considering the formal verification of two systolic algorithms for computing the convolution function. These detailed case studies were carried out within the CSDM software development system and in particular, uses the algebraic specification language SPECTRUM (see Broy et al [1993a, 1993b]) and the generic theorem prover Isabelle (see Paulson [1994]) with the higher-order object logic HOLCF (see Regensburger [1994]). Thus the case studies also demonstrate how existing software tools can be integrated into the formal design process.

For the development of large complex computing systems we need a methodology for stepwise design. Our simple verification methodology can be seen as representing a single step in such a development process and we begin to consider how to extend it to a full stepwise development methodology, by splitting requirements and iterating the verification process, in our concluding remarks presented in Section 5.

The structure of this paper is as follows. In Section 2 we introduce the specification language and software tools which will be used in the verification case study presented in this paper. We begin with a brief introduction to higher-order algebra and then introduce the algebraic specification language SPECTRUM. Next we introduce the formal software development environment CSDM and conclude the section with an overview of the generic theorem prover Isabelle and the object logic HOLCF. In Section 3 we present a simple methodology for the formal verification of software and hardware systems which is based on the concept of functional refinement. In Section 4 we demonstrate this verification methodology by considering the formal verification of two systolic algorithms for computing the convolution function. Finally in Section 5 we discuss the verification methodology we have proposed and consider its extension to a full development methodology for the stepwise design of computer systems. In the appendix section at the end of the paper we present the Isabelle HOLCF theories which were automatically generated from the SPECTRUM specifications used in the case studies presented in Section 4.
2 Background.

In this section we give the background for the case studies presented in Section 4 by introducing the specification language and software tools we will use. The case studies were originally performed within the framework of higher-order algebra (see Meineke and Steggle [1994]) and thus to aid the readers understanding of the specifications we will use we begin with a brief introduction to higher-order algebra. We then describe the CSDM (Correct Software Development Munich) software support environment in which our case studies will be developed. Next we introduce the algebraic specification language SPECTRUM which is used to formulate all specifications in the sequel. Finally, we give an overview of the generic theorem prover Isabelle and the object logic HOLCF which will be used to perform the verification proofs in our case studies.

2.1 Higher–Order Algebra.

Many-sorted first order algebraic methods are widely recognized as a practical and theoretically well founded formalism that can support many aspects of algorithm design such as specification, verification, transformation and refinement. The literature on these methods is extensive; recent surveys are Wirsing [1990], Ehrich [1994] and Meineke and Tucker [1993], while an introduction to the methods can be found in Ehren and Mahr [1985, 1990]. However many algebras arising in practice, for example algebras of streams used in hardware specification, are more naturally modelled as higher–order algebras. By a higher–order algebra we mean a many–sorted universal algebra in which the carrier sets may contain higher–order objects such as functions, functions acting on functions, etc. Higher–order algebraic methods have been shown to be substantially more expressive than first–order methods (see Kosluzenko and Meineke [1994] and Meineke [1995]). Thus there arises both a practical motivation and a need on fundamental theoretical grounds to consider higher–order algebraic methods as a formalism for algorithm design.

For an introduction to higher–order algebraic methods see for example Möller [1987], Möller et al [1988] and Meineke [1992a, 1992b]. For examples of their applications see Meineke and Steggle [1994], Meineke [1994] and Steggle [1995].

2.2 The Algebraic Specification Language SPECTRUM.

The specification language SPECTRUM is an algebraic specification language (see for example Wirsing [1990]) which provides a range of specification building operators, such as enrichment, hiding and parameterisation, to support specification in the large. SPECTRUM is based on a loose semantics, as opposed to the more popular approach of initial algebra semantics. Thus the axioms of a SPECTRUM specification can be full predicate logic formulas (over higher–order functions) allowing very abstract non–constructive requirement specifications to be written. In contrast to most conventional algebraic specification languages SPECTRUM was explicitly designed to support partial functions (based on the theory presented in Broy and Wirsing [1982]) and is based on a three valued logic (see Regensburger [1994]). Since SPECTRUM is oriented towards applicative programming a number of functional programming concepts have been included in the language, such as higher–order functions and the denotation of functions by λ-abstraction. The sort system of SPECTRUM allows parametric polymorphism and is based on the concept of sort classes, as found for example in the theorem prover Isabelle (see Paulson [1994]) and the functional programming language HASKEL (see Hudak et al [1992]).

For a detailed introduction to SPECTRUM we recommend Broy et al [1993a, 1993b]. The semantics of SPECTRUM is discussed in detail in Grosu and Regensburger [1994a, 1994b] and the associated proof system is discussed in Regensburger [1994]. For examples of the use of the
specification language SPECTRUM we refer the reader to the case studies Hettler et al [1994] and Slotosch [1995].

To illustrate the syntax of specifications in SPECTRUM we present two specifications which are required in the case studies presented in Section 4. The first specification \texttt{NAT} is a simple specification of the natural numbers.

\begin{verbatim}
NAT = {
  sort nat::EQ;    -- Declares a new sort of type class EQ --
  0 : nat;         -- Declares a constant of type nat --
  succ : nat -> nat; -- Declares a binary op from nat to nat --
  succ strict total; -- succ is both a strict and total op --
  nat freely generated by 0, succ;
}
\end{verbatim}

All text in a SPECTRUM specification following a \texttt{--} is a comment. The above specification begins by declaring a new sort \texttt{nat} which belongs to the type class \texttt{EQ}. This is the type class of sorts which have a decidable equality predicate and a proof is required to show that \texttt{nat} is indeed a member of this class (the proof is straightforward and so for brevity is omitted). This additional information about the sort \texttt{nat} is needed in Section 4 for the proof of the refinement condition for the programmable convolver (see Lemma 4.4.7). The specification also declares a constant symbol 0 of sort \texttt{nat} and a function symbol \texttt{succ} of type \texttt{nat \rightarrow nat}. The statement

\texttt{nat freely generated by 0, succ;} indicates that all elements of the sort \texttt{nat} are constructed from 0 and \texttt{succ}, and that the operation \texttt{succ} is injective. Omitting the word “\texttt{freely}” in the above statement would remove the requirement that \texttt{succ} be injective.

The second specification \texttt{RING} specifies a ring with unity (see for example Cohn [1965]).

\begin{verbatim}
RING = {
  sort ring;
  zero, one : ring;
  minus : ring \rightarrow ring;
  add : ring \times ring \rightarrow ring;
  mult : ring \times ring \rightarrow ring;
  minus, add, mult strict total;
  ring generated by zero, one, minus, add;
  axioms\ ALL x, y, z: ring in
\end{verbatim}
-- Axioms for ring with unity --

{comm}  add(x, y) = add(y, x);
{assocAD} add(x, add(y, z)) = add(add(x, y), z);
{assocMULT} mult(x, mult(y, z)) = mult(mult(x, y), z);
{idADD} add(x, zero) = x;
{idMULT1} mult(x, one) = x;
{idMULT2} mult(one, x) = x;
{unit}  add(x, minus(x)) = zero;
{dist1}  mult(x, add(y, z)) = add(mult(x, y), mult(x, z));
{dist2}  mult(add(x, y), z) = add(mult(x, z), mult(y, z));

endaxioms;

Given two specifications SPEC1 and SPEC2 we let SPEC1 + SPEC2 represent the specification formed by combining (i.e. taking the union) of the two specifications. Specifications can also be built hierarchically by enriching existing specifications. Given a specification SPEC we can enrich it to produce a specification SPEC' as follows.

SPEC' = { enrich SPEC;
  -- New sorts
  -- New constant and function symbols
  -- New axioms
}

When manipulating SPECTRUM specifications it is useful to be able to extract the signature and axioms of a specification. Let SPEC be a SPECTRUM specification. Then in the sequel we let Sig(SPEC) denote the signature of SPEC and Arms(SPEC) denote the axioms of SPEC. We denote by Sem(SPEC) the class of all models which satisfies the specification SPEC (i.e. loose semantics). Given a signature \( \Sigma \subseteq Sig(SPEC) \) we let

\[ Sem(SPEC)|_{\Sigma} = \{ A|_{\Sigma} \mid A \in Sem(SPEC) \}, \]

where \( A|_{\Sigma} \) represents the SPEC algebra \( A \in Sem(SPEC) \) restricted (i.e. reducted) to the signature \( \Sigma \).

2.3 CSDM Support System.

The CSDM (Correct Software Development Munich) software development support system was developed as part of the CSDM project for correct software development at the LMU Munich. It provides an environment in which the various stages of the development process can be integrated and managed. The system is based on the concept of development graphs, that is graphs in which nodes represent objects or components, such as specifications, programs and other documentation generated during the development process, and edges represent relationships between nodes, such as enrichment and refinement of specifications. The system was designed to allow for the reuse of
components and to this end a repository has been incorporated into the system. The repository is based on a functional database which allows for the expedient retrieval of components and provides a number of context sensitive search facilities. For an introduction to the CSDM system we refer the interested reader to Stahl and Gastinger [1996].

The CSDM system is not tied to any one specification language and can be instantiated to different formalisms. In the SPECTRUM instantiation the CSDM system provides tools for parsing specifications, for displaying the dependencies between hierarchical specifications, for calculating the declared and referenced signatures of specifications, and for the translation of specifications to functional ML and Isabelle HOLCF theories. The case studies we present in Section 4 were carried out within the SPECTRUM instantiation of the CSDM support system.

2.4 The Theorem Prover Isabelle and HOLCF.

Isabelle is a generic theorem which uses a fragment of higher-order logic to represent a variety of formal logics. It was developed for interactively reasoning about a variety of logics, for example first-order logic, higher-order logic, Zermelo-Fraenkel set theory and constructive type theory. Isabelle provides a higher-order meta logic based on the simply typed $\lambda$-calculus and the user interface is provided through the meta language Standard ML. Inference rules in Isabelle are generalised Horn clauses and rules are joined together in proofs by resolving such clauses. A simplifier is also provided by Isabelle which allows proofs to be performed using term rewriting. Isabelle can be seen as a proof manager and it provides support for handling multiple subgoals and proof decomposition. For an introduction to the theorem prover Isabelle we recommend Paulson [1990, 1994] and Nipkow [1989].

HOLCF (higher-order logic of computable functions) is an object logic for Isabelle which is based on the variant HOLCF which extends the well-known higher-order logic HOL with type classes. It provides a natural logical framework in which to reason about SPECTRUM specifications and tools exist for automatically converting SPECTRUM specifications into HOLCF theories. For a detailed description of the object logic HOLCF see Regensburger [1994].

3 Verification Methodology.

In this section we propose a simple methodology for the formal verification of software and hardware systems using algebraic specification methods. Our verification methodology is based on the notion of functional refinement and consists of constructing two specifications: a requirement specification which abstractly specifies the desired system and contains a distinguished environment or frame part; and a design specification which specifies a proposed implementation for the system. In order to verify that the design specification is a correct functional refinement of the requirement specification we first augment (i.e. instantiate) the design specification with the environment information contained within the requirement specification. This approach helps to facilitate the reuse of design specifications. We then show that this new theory satisfies the so-called consistency, preservation and refinement conditions which we define below. We demonstrate our verification methodology in the next section by considering the formal verification of the correctness of two systolic algorithms for computing the convolution function. Despite the simplicity of our verification methodology it turns out that it is sufficient for verifying a large class of computing systems. For example it can be used to verify examples of synchronous concurrent algorithms (see for example Thompson and Tucker [1994]) as illustrated by the case studies we present.

The verification methodology we propose consists of three phases, the so-called requirement specification, design specification and verification phases. A pictorial representation, based on
KORSO development graphs (see Wirsing [1992]), of the relationships between the specifications constructed during these three phases of the verification methodology is presented in Figure 1. In this figure the boxes represent axiomatic specifications, the single arrows indicate that the specification at the head of the arrow is included in the specification at the tail of the arrow and the double arrow denotes that the axioms in the specification at the head of the arrow are valid in all models of the specification at the tail.

In the first phase, the so called requirement specification phase, an abstract (often non-constructive) requirement specification is constructed for the desired system. We begin by defining the so called statement specification, denoted \textit{STAT SPEC}, which simply defines the name and type of all the operations involved in the system (thus the statement specification contains no axioms). In other words the statement specification declares the operations which are to be implemented and for this reason will be included in both the requirement and the design specifications. Note that since in our simple methodology we do not consider the implementation of the types given in the statement specification it would be more appropriate to use a partial signature in the statement specification. However, for simplicity and uniformity we use a full signature here when defining the statement specification. We enrich the statement specification to construct the requirement specification, denoted \textit{REQ SPEC}, which abstractly axiomatises the operations declared in the statement specification. In order to do this the requirement specification may need to use auxiliary types and operations which are not part of the desired system and we define a subspecification of the requirement specification, called the frame specification, for these auxiliary components. In fact given a statement and requirement specification we can always derive the frame specification; it simply consists of all operations in the requirement specification which are not in the statement specification, and all axioms of the requirement specification which do not contain operation symbols from the statement specification. The frame specification is needed to distinguish between those parts of the requirement specification which actually specify the system operations and those which specify the environment information (i.e. the auxiliary types and operations). This proves to be important in the verification phase of our methodology.

In the design specification phase we specify a design (or implementation) of the desired system by constructing a design specification, denoted \textit{DES SPEC}. The idea is that the design specification represents a functional refinement of the requirement specification with respect to the
operations in the statement specification and is thus an enrichment of the statement specification. We note that the design specification can be viewed as a complete specification in the sense that we do not identify its environment information, as we did for the requirement specification, since this is not needed for the verification phase.

The final phase is the verification phase in which we verify that the design specification is a correct functional refinement of the requirement specification, with respect to the statement specification. In order to do this we need to augment the design specification with the environment information used in the requirement specification and this leads us to define a verification specification, denoted VER\_SPEC, by combining the frame and design specifications (i.e. VER\_SPEC = FRAME\_SPEC \+ DES\_SPEC). Clearly, we have $\text{Sig}(\text{REQ}\_\text{SPEC}) \subseteq \text{Sig}(\text{VER}\_\text{SPEC})$ which is a prerequisite for the verification proof. This approach means that a design specification is not tied to a single requirement specification and thus facilitates the reuse of design specifications (which is essential for any practical formal design method). To prove that the design specification is a correct functional refinement of the requirement specification with respect to the statement specification, we need to show that three conditions, the so-called consistency, preservation and refinement conditions are satisfied. The consistency condition simply states that all specifications, i.e., the frame, requirement, design and verification specifications, must be consistent (where a specification is consistent if, and only if, there exists at least one non-unit model of the specification). The preservation condition ensures that the frame and design specifications can be consistently combined to produce a verification specification which preserves the semantics of the frame specification. It requires that the semantics of the frame specification is contained within the semantics of the design specification with respect to their intersecting types and operations. This condition corresponds to the idea that the frame specification can provide additional environment specific information to the operations in the design specification. This allows design specifications to abstract away from specific environment details and thus further facilitates the reuse of design specifications. It is important to note that due to this approach the design specification is only verified to be a correct functional refinement in the context of the current frame specification. The final condition, the refinement condition, states that the design specification must be a refinement (or implementation) of the requirement specification. That is, all axioms of the requirement specification must be valid in all models (i.e. the semantics) of the verification specification.

We formally define the notion of correct functional refinement as follows.

3.1 Definition. We say that the design specification DES\_SPEC is a correct functional refinement of the requirement specification REQ\_SPEC with respect to the statement specification STAT\_SPEC if, and only if, the following three conditions hold.

(1) Consistency Condition.
We need to ensure that all specifications in the development process are consistent. It suffices to show that the requirement and design specifications are consistent. The consistency of the frame specification then follows from the fact that it is a subspecification of the requirement specification. The consistency of the verification specification follows from the consistency of the frame and design specifications and the preservation condition below.

(2) Preservation Condition.
We need to ensure that the frame and design specifications can be consistently combined and furthermore, that the semantics of the frame specification is preserved by the resulting verification specification. It suffices to show that the semantics of the frame specification with respect to those operations which occur in both the frame and design specifications is preserved in the
semantics of the design specification, i.e.

\[ \text{Sem}(\text{FRAME SPEC}) \subseteq \text{Sem}(\text{DES SPEC}) \]

where \( \Sigma = \text{Sig}(\text{FRAME SPEC}) \cap \text{Sig}(\text{DES SPEC}) \).

(3) Refinement Condition.

Finally, we need to show that all the axioms of the requirement specification are valid in all models of the verification theory. By the definition of \( \text{VER SPEC} \) it suffices to show that

\[ \text{Sem}(\text{VER SPEC}) \models \text{Axms}(\text{REQ SPEC}) - \text{Axms}(\text{FRAME SPEC}) \].

Thus a verification proof will consist of showing that the above consistency, preservation and refinement conditions hold. Let us consider how we go about proving each condition. The consistency condition is proved by constructing a non-unit model of the requirement specification and a non-unit model of the design specification. In general this is straightforward to do using the semantic model that the requirement specification was based on and the implementation that the design specifications was based on. We note that another possible approach is to construct specifications which are known to be consistent by 

conservatively extending \( \) existing consistent specifications (we refer the interested reader to Regensburger [1994]). To prove that the preservation condition holds we need to show that for each \( A \in \text{Sem}(\text{FRAME SPEC}) \) there exists \( B \in \text{Sem}(\text{DES SPEC}) \) such that \( A \|_2 \equiv B \|_2 \). One approach to doing this is to choose an arbitrary \( A \in \text{Sem}(\text{FRAME SPEC}) \) and to then construct a \( \text{DES SPEC} \) model based on \( A \|_2 \). This approach will be used in the case studies presented in the next section. To show that the refinement condition holds it suffices to use a sound proof system to show that

\[ \text{Axms}(\text{VER SPEC}) \vdash \text{Axms}(\text{REQ SPEC}) - \text{Axms}(\text{FRAME SPEC}) \],

where \( \vdash \) denotes the inference relation for the chosen proof system. This has the added advantage of allowing tools from automated reasoning, such as term rewriting systems and theorem provers, to be used. In the case studies that follow we make use of the interactive theorem prover Isabelle and the logic HOLCF to prove the refinement condition.

The verification methodology we have presented can be seen as describing a single step in a formal development process in which functional refinement is used to develop a correct system design. We discuss this idea further in the concluding remarks presented in Section 5.

4 Case Study: Convolution.

In this section we apply the verification methodology presented in the previous section to the formal verification of two systolic algorithms, first introduced in Kung [1982], for computing the convolution function. The two case studies are based on Meinke and Steggles [1994] in which the framework of higher-order algebra was used to verify the systolic algorithms. Here we develop the case studies using the CSDM support system (see Section 2.2) and in particular, we use the specification language SPECTRUM (see Section 2.3) and the interactive theorem prover Isabelle (see Section 2.4).

We begin with an informal definition of the convolution function as a black box stream transformer.
4.1 Informal Definition of the Convolution Function.

Let \( n \in \mathbb{N} \) be some arbitrarily chosen but fixed non-zero natural number and let \( \mathbb{Z} = (\mathbb{Z} \cup \{0\}, +, \cdot, -, 0, 1) \) denote the ring of integers with unity. We can view convolution of sample size \( n \) over \( \mathbb{Z} \) as a black box, depicted in Figure 2, which takes as input a tuple of weights \( w = (w_1, \ldots, w_n) \in \mathbb{Z}^n \) and an infinite stream of integers \( a(0), a(1), a(2), \ldots \), and produces as output an infinite stream \( b(0), b(1), b(2), \ldots \) defined for each \( i \in \mathbb{N} \) by

\[
b(i) = (a(i) \times w_1) + \cdots + (a(i + n - 1) \times w_n).
\]

Thus, we can define convolution as a stream transformer (second-order function)

\[
\text{conv}^w : \mathbb{Z}^n \times [\mathbb{N} - \mathbb{Z}] \rightarrow [\mathbb{N} - \mathbb{Z}],
\]

defined for each \((w_1, \ldots, w_n) \in \mathbb{Z}^n, a \in [\mathbb{N} - \mathbb{Z}]\) and \( t \in \mathbb{N} \) by

\[
\text{conv}^w((w_1, \ldots, w_n), a)(t) = (a(t) \times w_1) + \cdots + (a(t + n - 1) \times w_n).
\]

This abstract definition of the convolution function will be used to formulate the requirement specification for convolution in the case studies that follow.

4.2 Synchronous Concurrent Algorithms.

The two systolic algorithms for convolution we will consider are examples of synchronous concurrent algorithms (SCAs) and we use the notation and terminology of SCA theory (see for example Thompson and Tucker [1994]) in presenting these algorithms. In this subsection we present a brief introduction to the theory of SCAs.

A synchronous concurrent algorithm (SCA) consists of a network of processing elements, called modules, which communicate via interconnections called channels. We refer to the network of an SCA as its architecture. All modules compute and communicate in parallel and these actions are synchronised to a global clock \( T = \{0, 1, 2, \ldots\} \). The network computes over a data type modelled as an algebra \( A \). Each processor module \( M_i \) performs a single computation which we specify by means of a total function \( f_i : A^{k(i)} \rightarrow A \) over the algebra \( A \), where module \( M_i \) has \( k(i) \) input channels and one output channel. Communication channels are allowed to finitely branch but they may not merge. Data enters the network at distinguished modules called sources and leaves at distinguished modules called sinks. In the case that \( A \) is a many-sorted algebra the modules and channels of the network are strongly typed by the sort set \( S \).

The operational semantics of an SCA can be described for each clock cycle \( t \in T \) as follows. Initially, for \( t = 0 \), each module places a predefined initial value on its output channel. At each subsequent clock cycle, \( t + 1 \), each processor module \( M_i \) applies its associated function \( f_i \) to the data \( a_{i1}, \ldots, a_{ik(i)} \) on its input channels at time \( t \) and places the result \( f_i(a_{i1}, \ldots, a_{ik(i)}) \) on its output channel at time \( t + 1 \). Each source module reads in a single input datum \( a(t) \in A \) from an input
stream $\alpha : T \rightarrow A$. Each sink module delivers a single output datum $\beta(t) \in A$ to an output stream $\beta : T \rightarrow A$. Thus, the behaviour of the entire network may be described by a system of simultaneous primitive recursive functions which we refer to as value functions.

An SCA processes infinite streams of input data and produces infinite streams of output data. Thus, we may describe its black box behaviour in terms of a second-order transformation on streams $Net : A^k \times [T \rightarrow A]^p \rightarrow [T \rightarrow A]^q$ called the network function, where $k$ is the total number of modules, $p$ is the number of source modules, and $q$ the number of sink modules. Given an initial state $\pi \in A^k$ for the entire network and $p$ input streams $\alpha_1, \ldots, \alpha_p : T \rightarrow A$ and initial state $\pi \in A^k$ then $Net(\pi, \alpha_1, \ldots, \alpha_p)$ defines the $q$ output streams $\beta_1, \ldots, \beta_q : T \rightarrow A$ produced by the network. The behaviour of each module $M_i$, for $i = 1, \ldots, k$, is defined by the value function

$$V^i : T \times [T \rightarrow A] \times A^k \rightarrow A,$$

where $V^i(t, \alpha_1, \ldots, \alpha_p, \pi)$ represents the output of module $i$ at time $t \in T$ on input streams $\alpha_1, \ldots, \alpha_p : T \rightarrow A$ and initial state $\pi \in A^k$.

To formalise the interaction of an SCA with its external environment we must specify input and output scheduling functions. These reconcile the system clock $T$ with some external environment clock by specifying when data is to be entered and retrieved from the network. The complete description of an SCA, in full detail, is obtained by combining the scheduling functions with the network specification.


4.3 The Non–Programmable Convolver.

The first systolic algorithm we consider for computing convolution has fixed or so called non–programmable weights and is thus restricted to computing a single convolution function. For this reason we refer to the algorithm as the non-programmable convolver. We begin with an informal description of the non–programmable convolver as an SCA and then formulate the statement, requirement and design specifications for the non–programmable convolver. We conclude by verifying that the design specification of the non–programmable convolver is a correct functional refinement of the requirement specification according to Definition 3.1.

We may describe the non–programmable convolver informally as follows.

4.3.1 Definition. The architecture of the non-programmable convolver is depicted in Figure 3. The function performed by each module $M_{ij}$ is defined as follows.

(i) Module $M_{1j}$, for $j = 1, \ldots, n$, computes the identity function $Id_Z : Z \rightarrow Z$, where $Id_Z(x) = x$.
(ii) Module $M_{2j}$, for $j = 1, \ldots, n$, returns the constant weight value $w_{(n+1)-j} \in Z$.
(iii) Module $M_{3j}$, for $j = 1, \ldots, n-1$, computes the ring inner product function $IP_Z : Z \times Z \times Z \rightarrow Z$, where $IP_Z(x, y, z) = (x \times_Z y) + z$.
(iv) Module $M_{4n}$ computes the ring multiplication operation $\times_Z$.

The input schedule $SCHED$ for this architecture loads data into the network during alternate
Figure 3: Architecture of the non-programmable convolver.

clock cycles and is defined by

\[ ISCHED : \mathbb{N} \rightarrow \mathbb{Z} \rightarrow \mathbb{N} \rightarrow \mathbb{Z}, \quad ISCHED(a)(t) = \begin{cases} 0, & \text{if } t \text{ is odd}, \\ a(t/2), & \text{otherwise}, \end{cases} \]

for \( a : \mathbb{N} \rightarrow \mathbb{Z} \) and \( t \in \mathbb{N} \). The output schedule \( OSCHED \) takes into account the initialisation period of the network, which is 2\( n \) clock cycles, and then retrieves data during alternate clock cycles. It is defined by

\[ OSCHED : \mathbb{N} \rightarrow \mathbb{Z} \rightarrow \mathbb{N} \rightarrow \mathbb{Z}, \quad OSCHED(a)(t) = a(2(n + t)), \]

for \( a : \mathbb{N} \rightarrow \mathbb{Z} \) and \( t \in \mathbb{N} \).

In the sequel we need to fix the sample size \( n \) to be an actual non-zero natural number. For convenience we take the sample size \( n = 3 \). The first step in the development process is to define the statement specification \( \text{STAT\_SIG\_FD} \). This will consist only of symbols for the fixed weights to be used in convolution and the convolution function.

4.3.2 Definition. Define the statement specification \( \text{STAT\_SIG\_FD} \) for convolution with fixed weights as follows.

\[
\text{STAT\_SIG\_FD} = \{
\begin{array}{l}
\text{sort nat;} \\
\text{sort ring;} \\
\text{w1, w2, w3 : ring;} \\
\text{Conv3 : (nat \rightarrow \text{ring}) \rightarrow (nat \rightarrow \text{ring});} \\
\text{Conv3 strict total;}
\end{array}
\}
\]
Next we formulate the requirement specification \texttt{REQ\_CONV\_FD} which is based on the abstract black box definition of convolution presented in Section 4.1. The requirement specification extends the statement specification \texttt{STAT\_SIG\_FD} and the specifications \texttt{NAT} and \texttt{RING} presented in Section 2.3.

4.3.3 Definition. Define the requirement specification \texttt{REQ\_CONV\_FD} for convolution with fixed weights as follows.

\texttt{REQ\_CONV\_FD = (enriches \texttt{NAT} \times \texttt{RING} \times \texttt{STAT\_SIG\_FD};}

\begin{verbatim}
axioms ALL t : nat, s : (nat -> ring) in
{Conv} Conv3(s)(t) = add(mult(w3, s(succ(succ(t)))),
add(mult(w2, s(succ(t))), mult(w1, s(t)) ) );
endaxioms;
\end{verbatim}

The frame specification simply consists of the union of the specifications \texttt{NAT} and \texttt{RING} and for this reason is not explicitly defined. We now formulate a design specification \texttt{NPC\_DES} for the non-programmable convolver based on Definition 4.3.1. This specification again extends the statement specification and the specifications \texttt{NAT} and \texttt{RING}.

4.3.4 Definition. Define the design specification \texttt{NPC\_DES} for the non-programmable convolver as follows.

\texttt{NPC\_DES = (enriches \texttt{NAT} \times \texttt{RING} \times \texttt{STAT\_SIG\_FD};

total;
strict;
y11, y12, y13, y21, y22, y23, y31, y32, y33 : ring;}

\begin{verbatim}
V11, V12, V13 : nat \times (nat -> ring) \times ring \times ring \times ring
\times ring \times ring \times ring \times ring \times ring \times ring;
V21, V22, V23 : nat \times (nat -> ring) \times ring \times ring \times ring
\times ring \times ring \times ring \times ring \times ring \times ring
\times ring \times ring \times ring \times ring \times ring \times ring;
V31, V32, V33 : nat \times (nat -> ring) \times ring \times ring \times ring
\times ring \times ring \times ring \times ring \times ring \times ring
\times ring \times ring \times ring \times ring \times ring \times ring;
Net : ring \times ring \times ring \times ring \times ring \times ring \times ring
\times ring \times (nat -> ring) \times (nat -> ring);
twice : nat -> nat;
\end{verbatim}
ISch : (nat -> ring) -> (nat -> ring);
OSch : (nat -> ring) -> (nat -> ring);

axioms  ALL t : nat, x11, x12, x13, x21, x22, x23, x31, x32, x33 : ring, s : (nat -> ring) in

-- Axioms to initialise modules --

{initV11}  V11(0,s,x11,x12,x13,x21,x22,x23,x31,x32,x33) = x11;
{initV12}  V12(0,s,x11,x12,x13,x21,x22,x23,x31,x32,x33) = x12;
{initV13}  V13(0,s,x11,x12,x13,x21,x22,x23,x31,x32,x33) = x13;
{initV21}  V21(0,s,x11,x12,x13,x21,x22,x23,x31,x32,x33) = x21;
{initV22}  V22(0,s,x11,x12,x13,x21,x22,x23,x31,x32,x33) = x22;
{initV23}  V23(0,s,x11,x12,x13,x21,x22,x23,x31,x32,x33) = x23;
{initV31}  V31(0,s,x11,x12,x13,x21,x22,x23,x31,x32,x33) = x31;
{initV32}  V32(0,s,x11,x12,x13,x21,x22,x23,x31,x32,x33) = x32;
{initV33}  V33(0,s,x11,x12,x13,x21,x22,x23,x31,x32,x33) = x33;

-- Axioms for modules function at time succ(t) --

{opV11}  V11(succ(t),s,x11,x12,x13,x21,x22,x23,x31,x32,x33) = s(t);
{opV12}  V12(succ(t),s,x11,x12,x13,x21,x22,x23,x31,x32,x33) = V11(t,s,x11,x12,x13,x21,x22,x23,x31,x32,x33);
{opV13}  V13(succ(t),s,x11,x12,x13,x21,x22,x23,x31,x32,x33) = V12(t,s,x11,x12,x13,x21,x22,x23,x31,x32,x33);
{opV21}  V21(succ(t),s,x11,x12,x13,x21,x22,x23,x31,x32,x33) = add(mult(V31(t,s,x11,x12,x13,x21,x22,x23,x31,x32,x33),
  V11(t,s,x11,x12,x13,x21,x22,x23,x31,x32,x33)),
  V22(t,s,x11,x12,x13,x21,x22,x23,x31,x32,x33));
{opV22}  V22(succ(t),s,x11,x12,x13,x21,x22,x23,x31,x32,x33) = add(mult(V32(t,s,x11,x12,x13,x21,x22,x23,x31,x32,x33),
  V12(t,s,x11,x12,x13,x21,x22,x23,x31,x32,x33)),
  V23(t,s,x11,x12,x13,x21,x22,x23,x31,x32,x33));
{opV23}  V23(succ(t),s,x11,x12,x13,x21,x22,x23,x31,x32,x33) = mult(V33(t,s,x11,x12,x13,x21,x22,x23,x31,x32,x33),
  V13(t,s,x11,x12,x13,x21,x22,x23,x31,x32,x33));
{opV31}  V31(succ(t),s,x11,x12,x13,x21,x22,x23,x31,x32,x33) = w3;
{opV32}  V32(succ(t),s,x11,x12,x13,x21,x22,x23,x31,x32,x33) = w2;
{opV33}  V33(succ(t),s,x11,x12,x13,x21,x22,x23,x31,x32,x33) = w1;

-- Axiom for Network function --

{Net}  Net(x11,x12,x13,x21,x22,x23,x31,x32,x33,s)(t) = V21(t,s,x11,x12,x13,x21,x22,x23,x31,x32,x33);

-- Axioms for twice --

{tw1}  twice(0) = 0;
{tw2} \quad \text{twice} (\text{succ}(t)) = \text{succ} (\text{succ} (\text{twice}(t))) ;

-- Axioms for Scheduling functions --

{insched1} \quad \text{ISch}(s)(\text{succ} (\text{twice}(t))) = \text{zero};

{insched2} \quad \text{ISch}(s)(\text{twice}(t)) = s(t);

{outsched} \quad \text{OSch}(s)(t) = \text{t} \cdot \text{twice} (\text{succ} (\text{succ} (\text{succ}(t)))) ;

-- Implementation of convolution function

{imp} \quad \text{Conv}3(s) = \text{OSch}(\text{Net}(y_{11}, y_{12}, y_{13}, y_{21}, y_{22}, y_{23}, y_{31}, y_{32}, y_{33}, \text{ISch}(s))) ;

\text{endaxioms} ;

\square

It turns out that the design specification already contains the frame specification \text{NAT + RING} and thus in this particular example there is no need to define a separate verification specification. The above specifications were loaded into the CSDM system and from within this system the specifications were parsed and the corresponding Isabelle HOLCF theories (presented in the appendix at the end of the paper) were automatically generated. During this process the CSDM system constructs a dependency graph, shown in Figure 4, depicting the relationships between the different specifications and theories. In this graph the rectangular boxes represent \text{SPEC-TRUM} specifications and the oval boxes represent Isabelle HOLCF theories. The arrows have the following meaning: the dashed arrows represent the inclusion of one specification in another; the wavy arrows represent the relationship between a specification and its Isabelle HOLCF theory; and the double arrow represents that the design specification is a correct refinement (according to Definition 3.1) of the requirement specification (which has still to be proved). It remains to show that the design specification \text{NPC_DES} is a correct functional refinement of the requirement specification \text{REQ_CONV_FD} with respect to the statement specification \text{STAT_SIG_FD}. According to Definition 3.1 we must show that the consistency, preservation and refinement conditions hold for our specifications.

We begin by proving that the requirement and design specifications are consistent.

4.3.5 Lemma. (Consistency) \text{The specifications REQ_CONV_FD and NPC_DES are consistent specifications.}

Proof. We need to construct a non-unit model of the requirement specification and a non-unit model of the design specification. This is straightforward to do using the informal definition of convolution given in Section 4.1 for the requirement specification and the informal definition of the non-programmable convolver given in Definition 4.3.1 for the design specification. \square

Next we prove that the design specification preserves the semantics of the frame specification.

4.3.6 Lemma. (Preservation) \text{The semantics of the frame specification \text{NAT+RING} is preserved by the design specification \text{NPC_DES}, i.e.}

\text{Sem(NAT + RING)} \subseteq \text{Sem(NPC_DES)} |_{\text{SIG(NAT+RING)}}
Figure 4: CSDM graph for the non-programmable convolver verification.

Proof. Given any $A \in \text{Sem}(\text{NAT} + \text{RING})$ we need to define an extension of $A$ to a NPC\_DES model $\tilde{A}$ such that $\tilde{A} = A|_{\text{Sem}(\text{NAT} + \text{RING})}$. This is straightforward to do using the fact that $\text{NAT} + \text{RING} \subseteq \text{NPC\_DES}$ and the informal definition of the non-programmable convolver given in Definition 4.3.1.

Finally we show that the design specification is a refinement of the requirement specification. Note that here the design specification plays the role of the verification specification.

4.3.7 Lemma. (Refinement) All models of the design specification NPC\_DES are also models of the requirement specification REQ\_CONV\_FD, i.e.

$$\text{Sem}(\text{NPC\_DES}) \models \text{Arms}(\text{REQ\_CONV\_FD}).$$

Proof. By the definition of NPC\_DES it suffices to show that

$$\text{Sem}(\text{NPC\_DES}) \models \text{Conv3}(s)(t) = \text{add}(\text{mult}(w3, s(\text{succ}(\text{succ}(t)))),$$

$$\text{add}(\text{mult}(w2, s(\text{succ}(t))), \text{mult}(w1, s(t))) ) .$$
We do this by proving that

\[
\begin{align*}
\text{Arms}(\text{NPC.DES}) \vdash \text{Conv3}(s)(t) &= \text{add}(\text{mult}(w_3, s)\text{succ}(\text{succ}(t))). \\
\text{add}(\text{mult}(w_2, s\text{succ}(t))), \text{mult}(w_1, s(t)))
\end{align*}
\]

using the theorem prover Isabelle with the object logic HOLCF and the HOLCF theories NAT.thy, RING.thy and NPC.DES.thy (see the Appendix) which were automatically generated from the corresponding SPECTRUM specifications by the CSDM system. The refinement proof is presented below.

use_thy NPC_DES;

open NPC_DES.thy

(* Derive axioms that initial values are defined *)
val yi1_Def = (NPC_DES_Def RS conjunct1);
val yi2_Def = (NPC_DES_Def RS conjunct2) RS conjunct1;
val yi3_Def = (((NPC_DES_Def RS conjunct2) RS conjunct2) RS conjunct1) RS conjunct2)
val y21_Def = (((NPC_DES_Def RS conjunct2) RS conjunct2) RS conjunct2) RS conjunct2)
val y22_Def = (((NPC_DES_Def RS conjunct2) RS conjunct2) RS conjunct2) RS conjunct2)
val y23_Def = (((NPC_DES_Def RS conjunct2) RS conjunct2) RS conjunct2) RS conjunct2)
val y31_Def = (((NPC_DES_Def RS conjunct2) RS conjunct2) RS conjunct2) RS conjunct2)
val y32_Def = (((NPC_DES_Def RS conjunct2) RS conjunct2) RS conjunct2) RS conjunct2)
val y33_Def = (((NPC_DES_Def RS conjunct2) RS conjunct2) RS conjunct2) RS conjunct2)

(* Derive axioms that functions are total *)
val Net_Total = (((((NPC_DES_Total RS conjunct2) RS conjunct2) RS conjunct2) RS conjunct2) RS conjunct2) RS conjunct2)
val twice_Total = (((((NPC_DES_Total RS conjunct2) RS conjunct2) RS conjunct2) RS conjunct2) RS conjunct2) RS conjunct2)
val Insched_Total = (((((NPC_DES_Total RS conjunct2) RS conjunct2) RS conjunct2) RS conjunct2) RS conjunct2) RS conjunct2)

(* Set up initial simpfile for rewriting *)
val def_ss = ccc1_ss addsimps [y11_Def, y12_Def, y13_Def, y21_Def, y22_Def, y23_Def, y31_Def, y32_Def, y33_Def, Net_Total, twice_Total, Insched_Total, NAT.NAT_Total];

(* **************************************************** *)
(* Lemma needed to push twice to outside *)
(* **************************************************** *)
val prems = goal NPC_DES.thy
  "t = UU =\=> succ[succ[twice[t]]] = twice[succ[t]]";

by (simp_tac (def_ss addsimps prems @ [tw2]) 1);
val rev_tw2 = result();

(* **************************************************** *)
(* Main goal to prove *)
(* **************************************************** *)
val prems = goal NPC_DES.thy
  "[is = UU; t = UU] =\=> Conv3[s][t] =
  add[mult[w3&s(succ[succ[t]])]] + add[mult[w2&s(succ[t])]] +
  mult[w1&s[t]]";

(* Set up simplfile for rewriting *)
val conv_es = def_ss addsimps prems @ [imp, Insched1, Insched2, Outsched, opV11, opV12, opV13, opV21, opV22, opV23, opV31, opV32, opV33, Net];

(* Begin rewriting with twice rule normal direction *)
by (simp_tac (conv_es addsimps [tw2]) 1);

(* Swap direction of twice rule, begin rewriting *)
by (simp_tac (conv_es addsimps [rev_tw2]) 1);

(* Check all subgoals have been proved *)
result();

We may now prove that the design specification for the non-programmable convolver is a correct functional refinement of the requirement specification.

4.3.8 Correctness Theorem. The design specification NPC_DES is a correct functional refinement of the requirement specification REQ_CONV_PD with respect to the statement specification STAT_SIG_PD.

Proof. Follows directly from Definition 3.1 and Lemmas 4.3.5, 4.3.6 and 4.3.7.

4.4 The Programmable Convolver.

The second systolic algorithm we consider is programmable and allows the user to load in the weights $w_1, \ldots, w_n \in \mathbb{Z}$ required for convolution as the initial values of modules $M_{3,n}, \ldots, M_{3,1}$.
respectively. The weight value \( w_i \) is then restored by a feedback channel to module \( M_{3,(n+1)-i} \) and remains constant throughout the computation.

Again we begin with an informal description of the architecture of the systolic algorithm, referred to in the sequel as the programmable convolver.

### 4.4.1 Definition

The architecture of the programmable convolver is depicted in Figure 5. The input and output schedules, \textit{ISCHED} and \textit{OSCHED}, and the function computed by each module \( M_{i,j} \) are the same as in Definition 4.3.1 except for the functions computed by modules \( M_{3,j} \), for \( j = 1, \ldots, n \). These are redefined as follows. Module \( M_{3,j} \), for \( j = 1, \ldots, n \), computes the identity function \( \text{Id}_{\mathbb{Z}} : \mathbb{Z} \rightarrow \mathbb{Z} \), where \( \text{Id}_{\mathbb{Z}}(x) = x \).

As before we begin the development process by defining a statement specification \texttt{STAT\_SIG} for the convolution function.

### 4.4.2 Definition

Define the statement specification \texttt{STAT\_SIG} for convolution as follows.

\[
\text{STAT\_SIG} = \{
\begin{align*}
\text{sort} & \text{ nat;} \\
\text{sort} & \text{ ring;} \\
\text{Conv3} & : \text{ring} * \text{ring} * \text{ring} * (\text{nat} \rightarrow \text{ring}) \rightarrow (\text{nat} \rightarrow \text{ring}); \\
\text{Conv3} & \text{ strict total};
\end{align*}
\]

Next we formulate the requirement specification \texttt{REQ\_CONV} which is based on the abstract black box definition of convolution presented in Section 4.1. Once again the requirement specification
extends the statement specification \texttt{STAT\_SIG} and the specifications \texttt{NAT} and \texttt{RING} presented in Section 2.3.

4.4.3 Definition. Define the requirement specification \texttt{REQ\_CONV} for convolution as follows.

\texttt{REQ\_CONV} = \{enriches \texttt{NAT} + \texttt{RING} + \texttt{STAT\_SIG};

axioms ALL x1, x2, x3 : ring, t : nat, s : (nat \rightarrow \texttt{ring}) in

\texttt{(Conv)} Conv3(x1,x2,x3,s)(t) = add(mult(x3, succ(succ(t)))),

add(mult(x2, succ(t))), mult(x1, s(t)));

endaxioms;

}\]

As before the frame specification simply consists of the union of the specifications \texttt{NAT} and \texttt{RING}, and for this reason is not explicitly defined. We now formulate a design specification \texttt{PC\_DES} for the programmable convolver based on Definition 4.4.1. This specification again extends the statement specification and the specifications \texttt{NAT} and \texttt{RING}.

4.4.4 Definition. Define the design specification \texttt{PC\_DES} for the programmable convolver as follows.

\texttt{PC\_DES} = \{enriches \texttt{NAT} + \texttt{RING} + \texttt{STAT\_SIG};

total;
strict;
y11, y12, y13, y21, y22, y23 : ring;

V11, V12, V13 : nat \rightarrow (nat \rightarrow \texttt{ring}) \rightarrow \texttt{ring} \rightarrow \texttt{ring} \rightarrow \texttt{ring}

* \texttt{ring} \rightarrow \texttt{ring} \rightarrow \texttt{ring} \rightarrow \texttt{ring} \rightarrow \texttt{ring};

V21, V22, V23 : nat \rightarrow (nat \rightarrow \texttt{ring}) \rightarrow \texttt{ring} \rightarrow \texttt{ring} \rightarrow \texttt{ring}

* \texttt{ring} \rightarrow \texttt{ring} \rightarrow \texttt{ring} \rightarrow \texttt{ring} \rightarrow \texttt{ring};

V31, V32, V33 : nat \rightarrow (nat \rightarrow \texttt{ring}) \rightarrow \texttt{ring} \rightarrow \texttt{ring} \rightarrow \texttt{ring}

* \texttt{ring} \rightarrow \texttt{ring} \rightarrow \texttt{ring} \rightarrow \texttt{ring} \rightarrow \texttt{ring};

Net : \texttt{ring} \rightarrow \texttt{ring} \rightarrow \texttt{ring} \rightarrow \texttt{ring} \rightarrow \texttt{ring}

* \texttt{ring} \rightarrow (nat \rightarrow \texttt{ring}) \rightarrow (nat \rightarrow \texttt{ring});

twice : nat \rightarrow nat;

ISch : (nat \rightarrow \texttt{ring}) \rightarrow (nat \rightarrow \texttt{ring});
OSch : (nat \rightarrow \texttt{ring}) \rightarrow (nat \rightarrow \texttt{ring});

axioms ALL t : nat, x11, x12, x13, x21, x22, x23, x31,
x32, x33 : ring, s : (nat -> ring) in

-- Axioms to initialise modules --
(iconv1) V11(0, s, x11, x12, x13, x21, x22, x23, x31, x32, x33) = x11;
(iconv2) V12(0, s, x11, x12, x13, x21, x22, x23, x31, x32, x33) = x12;
(iconv3) V13(0, s, x11, x12, x13, x21, x22, x23, x31, x32, x33) = x13;
(iconv21) V21(0, s, x11, x12, x13, x21, x22, x23, x31, x32, x33) = x21;
(iconv22) V22(0, s, x11, x12, x13, x21, x22, x23, x31, x32, x33) = x22;
(iconv23) V23(0, s, x11, x12, x13, x21, x22, x23, x31, x32, x33) = x23;
(iconv31) V31(0, s, x11, x12, x13, x21, x22, x23, x31, x32, x33) = x31;
(iconv32) V32(0, s, x11, x12, x13, x21, x22, x23, x31, x32, x33) = x32;
(iconv33) V33(0, s, x11, x12, x13, x21, x22, x23, x31, x32, x33) = x33;

-- Axioms for modules function at time succ(t) --
(opV11) V11(succ(t), s, x11, x12, x13, x21, x22, x23, x31, x32, x33) = s(t);
(opV12) V12(succ(t), s, x11, x12, x13, x21, x22, x23, x31, x32, x33) = V11(t, s, x11, x12, x13, x21, x22, x23, x31, x32, x33);
(opV13) V13(succ(t), s, x11, x12, x13, x21, x22, x23, x31, x32, x33) = V12(t, s, x11, x12, x13, x21, x22, x23, x31, x32, x33);
(opV21) V21(succ(t), s, x11, x12, x13, x21, x22, x23, x31, x32, x33) = add(mult(V31(t, s, x11, x12, x13, x21, x22, x23, x31, x32, x33),
V11(t, s, x11, x12, x13, x21, x22, x23, x31, x32, x33)),
V22(t, s, x11, x12, x13, x21, x22, x23, x31, x32, x33);
(opV22) V22(succ(t), s, x11, x12, x13, x21, x22, x23, x31, x32, x33) = add(mult(V32(t, s, x11, x12, x13, x21, x22, x23, x31, x32, x33),
V11(t, s, x11, x12, x13, x21, x22, x23, x31, x32, x33)),
V22(t, s, x11, x12, x13, x21, x22, x23, x31, x32, x33);
(opV23) V23(succ(t), s, x11, x12, x13, x21, x22, x23, x31, x32, x33) = mult(V33(t, s, x11, x12, x13, x21, x22, x23, x31, x32, x33),
V11(t, s, x11, x12, x13, x21, x22, x23, x31, x32, x33)),
V23(t, s, x11, x12, x13, x21, x22, x23, x31, x32, x33);
(opV31) V31(succ(t), s, x11, x12, x13, x21, x22, x23, x31, x32, x33) = V31(t, s, x11, x12, x13, x21, x22, x23, x31, x32, x33);
(opV32) V32(succ(t), s, x11, x12, x13, x21, x22, x23, x31, x32, x33) = V32(t, s, x11, x12, x13, x21, x22, x23, x31, x32, x33);
(opV33) V33(succ(t), s, x11, x12, x13, x21, x22, x23, x31, x32, x33) = V33(t, s, x11, x12, x13, x21, x22, x23, x31, x32, x33);

-- Axiom for Network function --

\{ \text{Net} \} \quad \text{Net}(x11, x12, x13, x21, x22, x23, x31, x32, x33, s)(t) = V21(t, s, x11, x12, x13, x21, x22, x23, x31, x32, x33);

-- Axioms for twice --

\{ \text{tw1} \} \quad \text{twice}(0) = 0;
\{ \text{tw2} \} \quad \text{twice}(\text{succ}(t)) = \text{succ}(\text{twice}(t));

-- Axioms for Scheduling functions --

\{ \text{Insched1} \} \quad \text{ISch}(s)(\text{twice}(t))) = \text{zero};
\{ \text{Insched2} \} \quad \text{ISch}(s)(\text{twice}(t)) = s(t);
\begin{verbatim}
 Outsched \[ OSch(s)(t) = \text{twice(succ(succ(succ(t))))}; \]

 -- Implementation of convolution function --
 (imp) \[ \text{Conv3}(x33,x32,x31,e) =
      OSch(\text{Net}(y11,y12,y13,y21,y22,y23,x31,x32,x33,ISch(s)));
\]

\end{verbatim}

Again the design specification already contains the frame specification \texttt{NAT + RING} and thus there is no need to define a separate verification specification. The above specifications were loaded in to the CSDM support system and from within this system the specifications were parsed and the corresponding Isabelle HOLCF theories were automatically generated. The dependency graph produced by the CSDM support system depicting the relationships between the different specifications and theories is shown in Figure 6. It remains to show that the design specification \texttt{PC\_DES} is a correct functional refinement of the requirement specification \texttt{REQ\_CONV} with respect to the statement specification \texttt{STAT\_SIG} (depicted by the double arrow in Figure 6). We begin by proving that the requirement and design specifications are consistent.

\textbf{4.4.5 Lemma.} (Consistency) The specifications \texttt{REQ\_CONV} and \texttt{PC\_DES} are consistent specifications.

\textbf{Proof.} We need to construct a non-unit model of the requirement specification and a non-unit model of the design specification. This is straightforward to do using the informal definition of convolution given in Section 4.1 and the informal definition of the programmable convolver given in Definition 4.4.1, and is left as an exercise for the reader.

Next we prove that the design specification preserves the semantics of the frame specification.

\textbf{4.4.6 Lemma.} (Preservation) The semantics of the frame specification \texttt{NAT\_RING} is preserved by the design specification \texttt{PC\_DES}, i.e.

\[ \text{Sem}(\texttt{NAT\_RING}) \subseteq \text{Sem}(\texttt{PC\_DES})\mid_{\text{spec}(\texttt{NAT\_RING})} \]

\textbf{Proof.} Given any \( A \in \text{Sem}(\texttt{NAT\_RING}) \) we need to define an extension of \( A \) to a \texttt{PC\_DES} model \( \bar{A} \) such that \( A = \bar{A} \mid_{\text{spec}(\texttt{NAT\_RING})} \). As before in Lemma 4.3.6 this is straightforward to do using the fact that \texttt{NAT\_RING} \( \subseteq \texttt{PC\_DES} \) and the informal definition of the programmable convolver given in Definition 4.4.1.

Finally we show that the design specification is a refinement of the requirement specification. Again the design specification \texttt{PC\_DES} plays the role of the verification specification since it already contains the frame specification \texttt{NAT\_RING}. Note that the refinement proof for the programmable convolver uses induction on the type \texttt{nat} to cope with the feedback loops associated with the weight modules.

\textbf{4.4.7 Lemma.} (Refinement) All models of the design specification \texttt{PC\_DES} are also models
Figure 6: CSDM graph for the programmable convolver verification.

of the requirement specification REQ_CONV, i.e.

$$\text{Sem}(\text{PC \_ DES}) \models \text{Axms}(\text{REQ \_ CONV}).$$

**Proof.** By the definition of PC\_DES it suffices to show that

$$\text{Sem}(\text{PC \_ DES}) \models \text{Conv}(x_1, x_2, x_3, s) = \text{add}(x_3, \text{succ}(\text{succ}(t))),$$

$$\text{add}(\text{mult}(x_2, \text{succ}(t)), \text{mult}(x_1, s(t))).$$

We do this by proving that

$$\text{Axms}(\text{PC \_ DES}) \vdash \text{Conv}(x_1, x_2, x_3, s) = \text{add}(x_3, \text{succ}(\text{succ}(t))),$$

$$\text{add}(\text{mult}(x_2, \text{succ}(t)), \text{mult}(x_1, s(t))).$$

using the theorem prover Isabelle with the object logic HOLCF and the HOLCF theories NAT.thy, RING.thy and PC\_DES.thy (presented in the appendix) which were automatically generated from the corresponding SPECTRUM specifications by the CSDM system. The refinement proof is presented below.

23
use.thy PC_DES;

open PC_DES.thy

(* Derive axioms that initial values are defined *)
val y11_Def = (PC_DES_Def RS conjunct1);
val y12_Def = (PC_DES_Def RS conjunct2) RS conjunct1;
val y13_Def = ((PC_DES_Def RS conjunct2) RS conjunct2) RS conjunct1;
val y21_Def = (((PC_DES_Def RS conjunct2) RS conjunct2) RS conjunct2)
  RS conjunct1;
val y22_Def = (((PC_DES_Def RS conjunct2) RS conjunct2) RS conjunct2)
  RS conjunct2 RS conjunct1;
val y23_Def = (((PC_DES_Def RS conjunct2) RS conjunct2) RS conjunct2)
  RS conjunct2 RS conjunct2 RS conjunct1;

(* Derive axioms that functions are total *)
val Net_Total = (((((((PC_DES_Total RS conjunct2) RS conjunct2) RS conjunct2)
  RS conjunct2) RS conjunct2) RS conjunct2) RS conjunct2 RS conjunct2)
  RS conjunct2 RS conjunct1;
val twice_Total = (((((((PC_DES_Total RS conjunct2) RS conjunct2) RS conjunct2)
  RS conjunct2) RS conjunct2) RS conjunct2) RS conjunct2 RS conjunct2)
  RS conjunct2 RS conjunct2 RS conjunct1;
val Insched_Total = (((((((PC_DES_Total RS conjunct2) RS conjunct2) RS conjunct2)
  RS conjunct2) RS conjunct2) RS conjunct2) RS conjunct2 RS conjunct2)
  RS conjunct2 RS conjunct2 RS conjunct2 RS conjunct1;

(* Set up initial simpfile for rewriting *)
val def_ss = ccc1_ss add_simps [y11_Def, y12_Def, y13_Def, y21_Def,
  y22_Def, y23_Def, Net_Total, twice_Total, Insched_Total,
  NAT.NAT_Total];

(* **********************************************)
(* Lemma needed to push twice to outside *)
(* **********************************************)
val prems = goal PC_DES.thy
  "t = UU ==> sucss[succ[twice[t]]] = twice[succ[t]]";
by (simp_tac (def_ss add_simps prems @ [tw2]) 1);
val rev_tw2 = result();

(* **********************************************)
(* Lemmas needed for Weight Modules *)
(* **********************************************)

(* Lemma: sucss is strict *)

24
goal PCDES.thy "succ[UU] = UU";
by (simp_tac (ccc1 ss addsimps [(Nat.NAT_Strict RS spec) RS mp]) 1);
val succ_Strict = result();

(* Lemma: succ is total *)
val prems = goal PCDES.thy "succ[x]=UU \rightarrow x=UU";
by (res_inst_tac ["x"] (Nat.nat.Gen RS spec) 1);
by (resolve_tac [impl] 4);
by (asm_simp_tac ccc1 ss 4);
by (resolve_tac [impl] 3);
by (simp_tac (ccc1 ss addsimps [(Nat.NAT_Def RS conjunct1)]) 3);
by (simp_tac (ccc1 ss addsimps [succ_Strict]) 2);
by (resolve_tac [adm_flat] 1);
by (resolve_tac [flat_lemma] 1);
val succ_Strict_Rev = (result() RS mp);

(* Main Lemmas for Weight Modules *)
(* i.e. V31(t, s, x11, ..., x33) = x3i *)
(* Lemma for V3i *)
val prems = goal PCDES.thy "[s=UU; x11=UU; x12=UU; x13=UU;
x21=UU; x22=UU; x23=UU; x31=UU; x32=UU; x33=UU] \rightarrow 
t'=UU \rightarrow V31[t#s#x11#x12#x13#x21#x22#x23#x31#x32#x33] = x31";

(* Apply generation rule for nats *)
by (resolve_tac [(Nat.nat.Gen RS spec)] 1);

(* Focus on UU case *)
by (simp_tac ccc1 ss 2);

(* Focus on Null Case *)
by (resolve_tac [impl] 2);
by (asm_simp_tac (ccc1 ss addsimps prems @ [initV31]) 2);

(* Focus on induction case *)
by (resolve_tac [impl] 2);
by (asm_simp_tac (ccc1 ss addsimps prems @
[opV31, succ_Strict_Rev]) 2);

(* Focus on Admissibility *)
by (resolve_tac [adm_flat] 1);
by (resolve_tac [flat_lemma] 1);

(* Label Lemma *)
val Lemma_V31 = (result() RS mp);

25
(* Focus on Admissibility *)
by (resolve_tac [adm_fat] 1);
by (resolve_tac [flat_lemma] 1);

(* Label Lemma *)
val Lemma_V33 = (result() RS mp);

(* ******************** *)
(* Main goal to prove *)
(* ******************** *)

val prems = goal PC_DES.thy
"[w1 := UU; w2 := UU; w3 := UU; t := UU] =>
Conv3[w1#w2#w3#s][t] = add[mult[w3#s[succ[succ[t]]]]#add[
  mult[w2#s[succ[t]]]#mult[w1#s[t]]]]";

(* Set up simpfile for rewriting *)
val conv_ss = def_ss addsimps prems @ [Imp, Insched1, Insched2,
  Outsched, opV11, opV12, opV13, opV21, opV22, opV23, Lemma_V31,
  Lemma_V32, Lemma_V33, Net];

(* Begin rewriting with twice rule normal direction *)
by (simp_tac (conv_ss addsimps [tw2]) 1);

(* Swap direction of twice rule, begin rewriting *)
by (simp_tac (conv_ss addsimps [rev_tw2]) 1);

(* Check all subgoals have been proved *)
result();

The use of induction in the above proof provided by the generation axiom nat.Gen (used in the theory NAT.thy to represent the statement "nat is freely generated by 0, succ" in the SPECTRUM specification NAT) requires some explanation. According to the generation axiom nat.Gen, in order to be able to use induction we need to show that the predicate (over the natural numbers) we are trying to prove is admissible (see for example Broy et al [1993b]). Since any predicate over a flat domain is admissible it suffices to show that nat is a flat domain. This is straightforward to do since we can show that nat is a member of the type class EQ of sorts with decidable equality predicates and there exists a lemma in HOLCF that states that all members of the type class EQ may be interpreted as flat domains.

We may now prove that the design specification for the programmable convolver is a correct functional refinement of the requirement specification.

4.4.8 Correctness Theorem. The design specification PC_DES is a correct functional refinement of the requirement specification REQ_CONV with respect to the statement specification STAT_SIG.
5 Conclusions.

In this paper we have presented a simple methodology for the formal verification of software and hardware systems using algebraic specification methods. This verification methodology was based on the concept of functional refinement and involved the construction of a statement specification which named the functions representing the desired system; a requirement specification which abstractly specified the functions named in the statement specification and contained a distinguished environment part referred to as the frame specification; and a design specification which specified a proposed implementation of the functions named in the statement specification. To verify that the design specification is a correct functional refinement of the requirement specification we first instantiated the design specification with the environment information contained in the requirement specification and then showed that the so-called consistency, preservation and refinement conditions were satisfied. The environment information contained within the requirement specification (given by the frame specification) was used to provide a context in which the implementation specified by the design specification was to be used. This approach ensures that a design specification is not tied to a single requirement specification and thus facilitates the reuse of design specifications. However, it is important to note that a design specification is verified correct only with respect to a particular environment.

We demonstrated our verification methodology by considering the formal verification of two systolic algorithms for computing convolution, the so-called non-programmable and programmable convolvers. The case studies were carried out within the CSDM development system and in particular, used the algebraic specification language SPECTRUM and the interactive theorem prover Isabelle with the higher-order logic HOLCF. Thus the case studies illustrated how existing software tools can be integrated into the formal verification process.

Despite its simplicity, the verification methodology we have presented is applicable to a surprisingly large class of computing systems. For example it can be used to verify examples of so-called synchronous concurrent algorithms as the case studies we presented illustrated. However, we acknowledge the limitations of our simple methodology. For example it excludes many recognized development methods, such as refinement by change of data structure (see for example Broy and Wirsing [1990]), and makes no provision for the stepwise design of computing systems. One possible approach is to view our verification methodology as a single step in a formal development process. Given a design specification we simply extract a group of new requirement specifications and apply our verification methodology again. We can then define a full development methodology based on iterating this process and then composing all the design specifications we have generated. Work on extending our verification methodology based on the above approach is underway and will be reported elsewhere.

Acknowledgements.

We are very grateful to O. Slotosch for his advice concerning SPECTRUM and HOLCF, and for his comments on draft versions of the paper. We would also like to thank R. Stabl, S. Gastinger and M. Berger for their help with the CSDM System. Finally, we acknowledge the financial support of the BNC MeDiGCS project during the preparation of this paper.
6 References.

Broy and Wirsing [1982],

Broy et al. [1993a],

Broy et al. [1993b],

Broy and Wirsing [1995],

Cohen et al. [1986],

Cohn [1965],

Ehrich [1994],

Ehrig and Mahr [1985],

Ehrig and Mahr [1990],

Ehrig and Weber [1985],

Eker and Tucker [1988],
Eker and Tucker [1989],

Goguen et al [1978],

Grosu and Regensburger [1994a],

Grosu and Regensburger [1994b],

Harman [1989],

Harman and Tucker [1988a],

Harman and Tucker [1988b],

Harman and Tucker [1994],

Henkin [1950],

Hettler et al [1994],

Hobley et al [1988],


Meinke [1992a],

Meinke [1992b],

Meinke [1994],

Meinke [1995],

Meinke and Tucker [1993],

Meinke and Steigles [1994],

Möller [1987],

Möller et al [1988],

Nipkow [1989],

Paulson [1990],

Paulson [1994],
Poole [1994],

Regensburger [1994],

Sannella and Wirsing [1983],

Slotosch [1995],

Stabl and Gastering [1996],

Steggles [1995],

Thompson [1987],

Thompson et al [1993],

Thompson and Tucker [1994],

Wirsing [1990],

Wirsing [1992],
7 Appendix.

In this appendix we present the HOLCF theories which were automatically generated from the SPECTRUM specifications for the two case studies presented in Section 4.

7.1 Base Theory Files.
Theory file NAT.thy

NAT = SPECHOLCF +

types

"nat" : 0

arities

"nat" : pcpo
"nat" : eq

consts

"null" : "nat"

"succ" : "nat -> nat"

"nat_When" : "x -> (nat -> 'a) -> nat -> 'a"

rules

nat_Gen  "|adm(P1); P1(UU); P1(null); \\ !y11. \\ ![ sucy11 = UU; P1(y11) ] >> \ \\ P1(succ[y11]) ] >> \ \\ (! x1.P1(x1))"

nat_Exh  "x = UU | x = null | (? y1. y1 = UU & x = succ[y1])"
nat.When 
  "(nat.When[f1][f2][UU] = UU) \n  \((\text{null } = \text{UU}) \rightarrow \text{nat.When}[f1][f2][\text{null}] = f1) \& \n  \((\text{succ}[x]) = \text{UU} \rightarrow \n  \text{nat.When}[f1][f2][\text{succ}[x]] = f2[x])"

NAT_Def "null = UU & succ = UU & nat.When = UU"

NAT_Strict "(! x1. (x1 = UU) \rightarrow \text{(succ}[x1] = UU))"

NAT_Total "(! x1. (x1 = UU) \rightarrow \text{(succ}[x1] = UU))"

end

Theory file RING.thy

RING = SPECHOLCF +

types
"ring" 0

arities
"ring" :: pcpo

consts
"zero" :: "ring"

"one._" :: "ring"

"minus._" :: "ring \rightarrow ring"

"add" :: "(ring \times ring) \rightarrow ring"

"mult" :: "(ring \times ring) \rightarrow ring"

rules

ring.Gen "[adm(P1); P1(UU); P1(zero); P1(one._); \n  \(!y31.\) \n  \[ ! \text{minus}[_y31] = \text{UU}; P1(y31) \]\) \[ \rightarrow \n  \] P1(minus[[_y31]]); \n  \) \rightarrow \n  \(!y41 y42.\)"
\ 
\ (| add[y1#y42] == \( UU; P1(y41); P1(y42) |) => \n\ P1(add[y1#y42]) => \n\ (! x1.\\/(x1))

\ring Exh \ "x = UU | x = zero | x = one. \ " \ 
\ (?! y1. y1 == UU & x = minus_[y1]) \ " \ 
\ (?! y1 y2. y1 == UU & y2 == UU & x = add[y1#y2])"

\comm \ "[x = UU; y == UU] => \ 
\ add[x#y] = add[y#x]"

\assADD \ "[x = UU; y == UU; z == UU] => \ 
\ add[x#add[y#z]] = add[add[x#y]#z]"

\assMULT \ "[x = UU; y == UU; z == UU] => \ 
\ mult[x#mult[y#z]] = mult[mult[x#y]#z]"

\identADD \ "x == UU => \ 
\ add[x#zero] = x"

\identMULT1 \ "x == UU => \ 
\ mult[x#one.] = x"

\identMULT2 \ "x == UU => \\
\ mult[one.#x] = x"

\unit \ "x == UU => \\
\ add[x#minus_[x]] = zero"

\dist1 \ "[x == UU; y == UU; z == UU] => \\
\ mult[x#add[y#z]] = add[mult[x#y]#mult[x#z]]"

\dist2 \ "[x == UU; y == UU; z == UU] => \\
\ mult[add[x#y]#z] = add[mult[x#z]#mult[y#z]]"

\RINGSDef \ "zero == UU & one. == UU & minus. == UU & add == UU & \mult == UU"

\RINGSStrict \ "(! x1. (x1 == UU) => (minus_[x1] == UU)) & \ 
\ (! x1 x2. (x1 == UU) | (x2 == UU) => (add[x1#x2] == UU)) & \\
\ & (! x1 x2. (x1 == UU) | (x2 == UU) => \\
\ (mult[x1#x2] == UU))"

\RINGSTotal \ "(! x1. (x1 == UU) => (minus_[x1] == UU)) & \\
\ (! x1 x2. (x1 == UU) & (x2 == UU) => \\
\ (add[x1#x2] == UU)) & (! x1 x2. (x1 == UU) & \\
\ (x2 == UU) => (mult[x1#x2] == UU))"
7.2 Theory Files for Non-Programmable Convolver.

Theory file STAT_SIG_FD.thy

\textit{STAT\_SIG\_FD} = \textit{SPECHOLCF} +

\begin{verbatim}
types
"nat_"  0
"ring"  0

arities
"nat_"  :: pcppo
"ring"  :: pcppo

consts
"w1"  :: "ring"
"w2"  :: "ring"
"w3"  :: "ring"
"Conv3"  :: "(nat_ \rightarrow ring) \rightarrow nat_ \rightarrow ring"

rules
\textit{STAT\_SIG\_FD\_Def}  "w1 = UU \& w2 = UU \& w3 = UU \& Conv3 = UU"
\textit{STAT\_SIG\_FD\_Strict} "(\forall x1. (x1 = UU) \rightarrow (Conv3[x1] = UU))"
\textit{STAT\_SIG\_FD\_Total} "(\forall x1. (x1 = UU) \rightarrow (Conv3[x1] = UU))"
\end{verbatim}

end

Theory file REQ_CONV_FD.thy

\textit{REQ\_CONV\_FD} = \textit{NAT} + \textit{RING} + \textit{STAT\_SIG\_FD} +
rules
Conv "([s = UU; t = UU]) => \nConv3[a][c] = \n add[mult[w3#s[succ[succ[t]]]]#add[mult[w2#s[\n succ[t]]]#mult[w1#s[t]])]"

end

Theory file DES_NPC.thy
NPC_DES = NAT + RING + STAT_SIG_FD +

consts
"y11" :: "ring"
"y12" :: "ring"
"y13" :: "ring"
"y21" :: "ring"
"y22" :: "ring"
"y23" :: "ring"
"y31" :: "ring"
"y32" :: "ring"
"y33" :: "ring"

"V11" :: "(nat_ + (nat_ => ring) * ring * ring * ring * ring * ring +" * ring * ring * ring * ring * ring) => ring"
"V12" :: "(nat_ + (nat_ => ring) * ring * ring * ring * ring * ring * ring * ring * ring * ring * ring) => ring"
"V13" :: "(nat_ + (nat_ => ring) * ring * ring * ring * ring * ring * ring * ring * ring * ring * ring) => ring"
"V21" :: "(nat_ + (nat_ => ring) * ring * ring * ring * ring * ring * ring * ring * ring * ring * ring) => ring"
"V22" :: "(nat_ + (nat_ => ring) * ring * ring * ring * ring * ring * ring * ring * ring * ring * ring) => ring"
7.3 Theory Files for Programmable Convolver.

Theory file STAT.SIG.thy

STAT_SIG = SPECHOLCF +

types
"nat" 0
"ring" 0

arities
"nat" :: pcpo
"ring" :: pcpo

consts
"Conv3" :: "(ring * ring * ring * nat -> ring) -> nat -> ring"

rules
STAT_SIG_Def "Conv3 "= UU"

STAT_SIG_Strict "\n\(! \ x1 \ x2 \ x3 \ x4. (x1 = UU) | (x2 = UU) | (x3 = UU) |\n\(x4 = UU) \rightarrow (\text{Conv3}[x1#x2#x3#x4] = UU))"

STAT_SIG_Total "\n\(! \ x1 \ x2 \ x3 \ x4. (x1 "= UU) \& (x2 "= UU) \& (x3 "= UU)\n\& (x4 "= UU) \rightarrow (\text{Conv3}[x1#x2#x3#x4] "= UU))"

end

Theory file REQ.CONV.thy

REQ_CONN = NAT + RING + STAT.SIG +

rules
Conv $\cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \
"V33" :: "(nat_ * (nat_ -> ring) * ring * ring * ring * ring
* ring * ring * ring * ring * ring) -> ring"

"Net" :: "((ring * ring * ring * ring * ring * ring
* ring * ring * (nat_ -> ring)) -> nat_ -> ring"

"twice" :: "nat_ -> nat_"

"Tsch" :: "(nat_ -> ring) -> nat_ -> ring"

"Gsch" :: "(nat_ -> ring) -> nat_ -> ring"

rules

initV11 "[s = UU;x11 = UU;x12 = UU;x13 = UU;x21 = UU:\n x22 = UU;x23 = UU;x31 = UU;x32 = UU;x33 = UU] ==\n V11[null##x11#x12#x13#x21#x22#x23#x31#x32#x33] = x11"

initV12 "[s = UU;x11 = UU;x12 = UU;x13 = UU;x21 = UU:\n x22 = UU;x23 = UU;x31 = UU;x32 = UU;x33 = UU] ==\n V12[null##x11#x12#x13#x21#x22#x23#x31#x32#x33] = x12"

initV13 "[s = UU;x11 = UU;x12 = UU;x13 = UU;x21 = UU:\n x22 = UU;x23 = UU;x31 = UU;x32 = UU;x33 = UU] ==\n V13[null##x11#x12#x13#x21#x22#x23#x31#x32#x33] = x13"

initV21 "[s = UU;x11 = UU;x12 = UU;x13 = UU;x21 = UU:\n x22 = UU;x23 = UU;x31 = UU;x32 = UU;x33 = UU] ==\n V21[null##x11#x12#x13#x21#x22#x23#x31#x32#x33] = x21"

initV22 "[s = UU;x11 = UU;x12 = UU;x13 = UU;x21 = UU:\n x22 = UU;x23 = UU;x31 = UU;x32 = UU;x33 = UU] ==\n V22[null##x11#x12#x13#x21#x22#x23#x31#x32#x33] = x22"

initV23 "[s = UU;x11 = UU;x12 = UU;x13 = UU;x21 = UU:\n x22 = UU;x23 = UU;x31 = UU;x32 = UU;x33 = UU] ==\n V23[null##x11#x12#x13#x21#x22#x23#x31#x32#x33] = x23"

initV31 "[s = UU;x11 = UU;x12 = UU;x13 = UU;x21 = UU:\n x22 = UU;x23 = UU;x31 = UU;x32 = UU;x33 = UU] ==\n V31[null##x11#x12#x13#x21#x22#x23#x31#x32#x33] = x31"

initV32 "[s = UU;x11 = UU;x12 = UU;x13 = UU;x21 = UU:\n x22 = UU;x23 = UU;x31 = UU;x32 = UU;x33 = UU] ==\n V32[null##x11#x12#x13#x21#x22#x23#x31#x32#x33] = x32"

initV33 "[s = UU;x11 = UU;x12 = UU;x13 = UU;x21 = UU;
(x6 = UU) | (x7 = UU) | (x8 = UU) | (x9 = UU) | \\
(x10 = UU) | (x11 = UU) --> \\
(V21[(x1 = x2 = x3 = x4 = x5 = x6 = x7 = x8 = x9 = x10 = x11) = UU]) & \\
(! x1 x2 x3 x4 x6 x7 x8 x9 x10 x11. (x1 = UU) | \\
(x2 = UU) | (x3 = UU) | (x4 = UU) | (x5 = UU) | \\
(x6 = UU) | (x7 = UU) | (x8 = UU) | (x9 = UU) | \\
(x10 = UU) | (x11 = UU) --> \\
(V22[(x1 = x2 = x3 = x4 = x5 = x6 = x7 = x8 = x9 = x10 = x11) = UU]) & \\
(! x1 x2 x3 x4 x6 x7 x8 x9 x10 x11. (x1 = UU) | \\
(x2 = UU) | (x3 = UU) | (x4 = UU) | (x5 = UU) | \\
(x6 = UU) | (x7 = UU) | (x8 = UU) | (x9 = UU) | \\
(x10 = UU) | (x11 = UU) --> \\
(V23[(x1 = x2 = x3 = x4 = x5 = x6 = x7 = x8 = x9 = x10 = x11) = UU]) & \\
(! x1 x2 x3 x4 x6 x7 x8 x9 x10 x11. (x1 = UU) | \\
(x2 = UU) | (x3 = UU) | (x4 = UU) | (x5 = UU) | \\
(x6 = UU) | (x7 = UU) | (x8 = UU) | (x9 = UU) | \\
(x10 = UU) | (x11 = UU) --> \\
(V33[(x1 = x2 = x3 = x4 = x5 = x6 = x7 = x8 = x9 = x10 = x11) = UU]) & \\
(! x1 x2 x3 x4 x5 x6 x7 x8 x9 x10. (x1 = UU) | \\
(x2 = UU) | (x3 = UU) | (x4 = UU) | (x5 = UU) | \\
(x6 = UU) | (x7 = UU) | (x8 = UU) | (x9 = UU) | \\
(x10 = UU) --> (Net[x1 = x2 = x3 = x4 = x5 = x6 = x7 = x8 = x9 = x10] = UU) & (! x1. (x1 = UU) --> (twice[x1] = UU)) & \\
(! x1. (x1 = UU) --> (ISch[x1] = UU)) & \\
(! x1. (x1 = UU) --> (OSch[x1] = UU))"
(x6 == UU) & (x7 == UU) & (x8 == UU) & (x9 == UU) &
(x10 == UU) & (x11 == UU) --> |
(V13[x1#2#3#4#5#6#7#8#9#10#11] == UU)) & |
(! x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11. (x1 == UU) &
(x2 == UU) & (x3 == UU) & (x4 == UU) & (x5 == UU) &
(x6 == UU) & (x7 == UU) & (x8 == UU) & (x9 == UU) &
(x10 == UU) & (x11 == UU) --> |
(V21[x1#2#3#4#5#6#7#8#9#10#11] == UU)) & |
(! x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11. (x1 == UU) &
(x2 == UU) & (x3 == UU) & (x4 == UU) & (x5 == UU) &
(x6 == UU) & (x7 == UU) & (x8 == UU) & (x9 == UU) &
(x10 == UU) & (x11 == UU) --> |
(V22[x1#2#3#4#5#6#7#8#9#10#11] == UU)) & |
(! x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11. (x1 == UU) &
(x2 == UU) & (x3 == UU) & (x4 == UU) & (x5 == UU) &
(x6 == UU) & (x7 == UU) & (x8 == UU) & (x9 == UU) &
(x10 == UU) & (x11 == UU) --> |
(V23[x1#2#3#4#5#6#7#8#9#10#11] == UU)) & |
(! x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11. (x1 == UU) &
(x2 == UU) & (x3 == UU) & (x4 == UU) & (x5 == UU) &
(x6 == UU) & (x7 == UU) & (x8 == UU) & (x9 == UU) &
(x10 == UU) & (x11 == UU) --> |
(V31[x1#2#3#4#5#6#7#8#9#10#11] == UU)) & |
(! x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11. (x1 == UU) &
(x2 == UU) & (x3 == UU) & (x4 == UU) & (x5 == UU) &
(x6 == UU) & (x7 == UU) & (x8 == UU) & (x9 == UU) &
(x10 == UU) & (x11 == UU) --> |
(V32[x1#2#3#4#5#6#7#8#9#10#11] == UU)) & |
(! x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11. (x1 == UU) &
(x2 == UU) & (x3 == UU) & (x4 == UU) & (x5 == UU) &
(x6 == UU) & (x7 == UU) & (x8 == UU) & (x9 == UU) &
(x10 == UU) & (x11 == UU) --> |
(V33[x1#2#3#4#5#6#7#8#9#10#11] == UU)) & |
(! x1 x2 x3 x4 x5 x6 x7 x8 x9 x10. (x1 == UU) &
(x2 == UU) & (x3 == UU) & (x4 == UU) & (x5 == UU) &
(x6 == UU) & (x7 == UU) & (x8 == UU) & (x9 == UU) &
(x10 == UU) --> (Not(x1#2#3#4#5#6#7#8#9#10)) & |
(! UU)) & (! x1. (x1 == UU) --> (twice[x1] == UU)) & 
(! x1. (x1 == UU) --> (IsSch[x1] == UU)) & 
(! x1. (x1 == UU) --> (OSh[x1] == UU))
end