Verification of Rewrite Based Specifications using Proof Assistants

Thomas Mailleux Sant'Ana[1*] **and Mauricio Ayala-Rincón**[1]

[1]Mestrado em Informática e Departamento de Matemática,
Universidade de Brasília, Brasília D. F., Brasil.

mailleux@gmx.net, ayala@mat.unb.br

Abstract. *Recent works point out the application of rewriting-logic environments for the specification of hardware. When these specification are proved to be correct one can additionally apply them for the simulation, testing and even analysis of the conceived specified hardware. But theorem proving mechanisms are not included as basic/natural components of rewriting-logic environments (such as* ELAN*, CafeObj and Maude). Even worst, they are not able to handle proofs guided by basic methods of rewriting theory. Consequently, the correctness of these specifications have been done by hand. In this work we present a new practical methodology, which is based on a semantically intelligent translation of rewriting-logic specifications in* ELAN *to theories in the specification language* PVS*(a well-known proof assistant). This translation includes generation of lemmas to be checked for guaranteeing the joinability of critical pairs of the rewriting rules of the original specification.*

Resumo. *Trabalhos recentes mostram como usar ambientes de reescrita lógica na especificação de hardware. As especificações, uma vez demonstradas corretas, podem ser simuladas, testadas e até analisadas no ambiente de reescrita. Entretanto sistemas de reescrita lógica (como* ELAN*, CafeObj e Maude) não incluem mecanismos naturais/básicos de prova de teoremas. Pior ainda, eles são incapazes de tratar provas orientadas pelos métodos básicos da teoria de reescrita. Por isto as provas de correção devem ser feitas de forma manual. Neste trabalho propomos uma metodologia prática, baseada da tradução semanticamente inteligente de especificações em reescrita lógica em* ELAN *para teorias na linguagem de especificação do* PVS*(um assistente de prova bem conhecido). Esta tradução inclui a geração de lemas a serem provados que garantem a juntabilidade dos pares críticos da especificação original em reescrita.*

1. Introduction

In [Ayala-Rincón et al., 2002] the correctness proof of the specification of a speculative processor, presented as a term rewriting system (TRS), was obtained showing that this processor imitates a basic RISC processor and vice-versa. The ELAN specification of this processor was given by a term rewriting system which consists of four sets of specialized rewriting rules: for issuing instructions to the buffer, load/store in memory instructions, branch prediction rules and arithmetic and propagation rules. These subset of rules coincide with the ones in [Arvind and Shen, 1999] except that in this rewriting-logical setting logical strategies were applied for obtaining a clean discrimination between parts of the processor as well as for implementing different speculative strategies. For proving that

[*]Corresponding author.

each processor imitates the other, it was necessary to verify the convergence of all rules except the issuing instruction rules of the speculative processor. This was done by proving Noetherianity of this set of rules and applying the Knuth-Bendix-Huet Critical Pair Lemma by hand. This involved exhaustive computation of critical pairs between rewriting rules and subsequent verification of their joinability.

The proposed methodology allows us to mechanically complete this task during the translation to the language of the proof assistant PVS. It is important to stress that the intelligence involved in this translation is not redundant, because no strategies for handling proof based in rewriting basic theory are available in PVS (as well as in most of the known practical proof assistants). Further proofs of integrity constraints of the target specifications are then done in PVS and in the case these proofs are not possible a dynamically reformulation of the rewriting based specification and continuous translation to the language of PVS is possible. In this way verification of complex specifications can be done quickly and safely.

The applicability of the proposed methodology is not restricted to the case of hardware specification, but it works for any rewrite based specification (as the example in the appendix illustrates). Specification via rewriting-logic has been showed of practical interest for the modeling and simulation of non standard hardware technologies (for which no commercial tool of correct synthesis is available) such as the innovative reconfigurable systems used in compact and portable systems [Ayala-Rincón et al., 2003, Ayala-Rincón et al., 2004]. Because of this we believe the proposed methodology will be of practical interest in the correct development of reconfigurable technologies.

The remainder of this section briefly discusses the required theoretical basis. Section 2 discusses the goals and outline of the methodology being proposed. Section 3 discusses the translation techniques and choices, it describes why certain routes where used. Section 4 presents one example of application for a large ELAN specification of a basic AX RISC processor. The last section concludes and presents future work. For the benefit of the review process we include an appendix with a small example together with full listings of the input ELAN specification and its translation to PVS.

1.1. Theoretical Basis

We include the minimal needed notions on rewriting. For detailed presentations see [Brader and Nipkow, 1998, van Rijsbergen, 2003].

Rewriting theory has been successfully applied into different areas of computer science as an abstract formalism for assisting the simulation, verification and deduction of complex computational objects and processes. In the context of computer architectures, rewriting theory has been applied as a tool for reasoning about hardware design. To review only a reduced set of different approaches in this direction, we mention the work of Kapur who has used his well-known Rewriting Rule Laboratory - RRL for verifying arithmetic circuits [Kapur and Subramaniam, 1997, Kapur and Subramaniam, 2000] as well as Arvind's group at MIT that treated the specification of processors over simple architectures, the rewrite-based description and synthesis of simple logical digital circuits and the description of cache protocols over memory systems [Arvind and Shen, 1999, Hoe and Arvind, 2004]. Rewriting-logic, that extends the pure rewriting paradigm allowing for logical control of the application of the rules by logic strategies, has been showed of greater flexibility than purely rewriting for discriminating between fixed and reconfigurable elements of the innovative reconfigurable architectures used in modern, portable and compact technologies. This allows for a natural and quick conception and simulation of implementations of emerging computing paradigms such as *configware* and *morphware* [Becker and Hartenstein, 2003] over the conceived reconfigurable architectures which includes the sophisticated case of dynamically reconfiguration. In particular, we have applied this specification methodology to the design and modeling of systems for efficient algebraic computations [Ayala-Rincón et al., 2003] and for reconfigurable systems for general dynamic programming based implementations of sequence processing methods such as sequence alignment and approximate pattern matching [Ayala-Rincón et al., 2004].

A Term Rewriting System (TRS) is defined as a triple $\langle R, S, S_0 \rangle$, where S and R are respectively sets of terms and of rewrite rules of the form $l \to r$ if $p(l)$ being l and r terms and p a predicate and where S_0 is the subset of allowed initial terms of S. l and r are called the left-hand and right-hand sides of the rule and $p(l)$ its condition. In the architectural context, terms and rules represent states and state transitions, respectively. A term s can be rewritten or reduced to the term t, denoted by $s \to t$, whenever there exists a subterm s' of s that can be transformed according to some rewrite rule into the term s'' such that replacing the occurrence of s' in s with s'' gives t. A term that cannot be rewritten is said to be in normal form. The relation over S given by the previous rewrite mechanism is called the rewrite relation of R and is denoted by $\to_R$. Its inverse is denoted by ${}_R\leftarrow$ and its reflexive-transitive closure by $\to_R^*$ and its equivalence closure by $\leftrightarrow_R^*$. The important notions of terminating and confluence properties are defined as usual. These notions correspond to the practical computational aspects as the determinism of processes and their finiteness. A TRS is said to be terminating if there are no infinite sequences of the form $s_0 \to s_1 \to \cdots$. a TRS is said to be confluent if for all divergence of the form $s \to t_1$, $s \to t_2$, there exists a term u such that $t_1 \to^* u^* \leftarrow t2$.

The use of the subset of initial terms S_0, representing possible initial states in the architectural context (which is not standard in rewriting theory), is simply to define what is a ”legal” state according to the set of rewrite rules R; i.e., t is a legal term (or state) whenever there exists an initial state $s \in S_0$ such that $s \to^* t$. Using these notions one can model the operational semantics of algebraic operators and functions. Although in the pure rewriting context rules are applied in a truly non deterministic manner, in the practice it is necessary to have the control of the ordering in which rules are applied. Thus, rewriting jointly with logic, that is known as rewriting-logic [Martí-Oliet and Meseguer, 2002], has been showed of practical applicability in this context of specification of processors since they may be adapted for discriminatingly representing in the necessary detail many hardware elements involved in processors [Ayala-Rincón et al., 2002, Ayala-Rincón et al., 2003, Ayala-Rincón et al., 2004].

Efforts in the implementation of computational environments and programming languages based on rewriting (matching and substitution) and logic include well-known tools such as ELAN, CafeObj and Maude (see [Martí-Oliet and Meseguer, 2002] for recent descriptions of these systems). These systems are useful for implementing and running specifications, but except for the treatment of types they don't include elaborated and natural tools for proving correctness of these specifications, correction that is based on rewriting basic theory. In particular, when specifying hardware systems one need to prove their confluence by applying highly used criteria such as the Knuth-Bendix-Huet Critical Pair Lemma. This lemma states that for a terminating rewriting systems one can check confluence whenever all divergences generated by overlapping of left-hand sides of the rules are joinable. Which states a local and effective criterion for verifying convergence of rewriting systems and is the basis of the well-known Knuth-Bendix completion procedure [Brader and Nipkow, 1998, van Rijsbergen, 2003].

The proposed methodology translates rewriting based specifications (in ELAN) to the language of the proof assistant PVS, which is the proof assistant used in our experiments. Advantages of PVS include the power of dependent types and higher order logic strategies [Owre et al., 1998, Rushby et al., 1998] as well as a lot of accumulated experience in its application in hardware verification [Owre et al., 1994], [Cyrluk et al., 1994], [R. Hosabettu and G. Gopalakrishnan and M. Srivas, 2003] and [Miller and Srivas, 1995]. Since PVS does not include strategies for rewriting theory, during the translation, critical pairs are generated and the corresponding (joinability) lemmas written in the language of PVS for further verification.

In conditional rewriting systems, critical pairs whose conditions are unsatisfiable are considered trivially joinable or simply omitted [Ohlebusch, 2002]. But our translation does not includes built-in decision algorithms for deciding whether these conditions are or not satisfiable. Consequently, trivial critical pairs are maintained and their omission is let for the PVS phase, where one can apply built-in decision procedures of this proof assistant.

In PVS type-checking makes part of the theorem proving framework, which makes it possible to prove type conditions that are central for correctness proofs of the specifications. *Predicate subtypes* express mathematical ideas such as naturals are contained in integers, constraints which determine whether a number is even, odd, prime, etc. Predicate subtypes generate a series of *Type Correctness Conditions*, called TCC in PVS, whose demonstrations depend on the extra logic embedded in these predicates instead on conventional type-checking.

PVS tries to prove all TCCs automatically and whenever it cannot do so, it generates a proof *obligation* for the user. Any lemma or theorem that depends on that obligation will be considered incomplete until that TCC is proved. This can assist users in detecting errors and problems. For example, consider the following lemma:

$$div\_cancel : \textbf{LEMMA}\ \forall x, y \in \mathbb{Z} : \frac{x \times y}{x} = x$$

This is not valid because of the possible division by zero. PVS checks all condition required for the lemma and proposes the following TCC:

$$div\_cancel\_TCC1 : \textbf{OBLIGATION}\ \forall(x) : x \neq 0;$$

This TCC cannot be proved, so any proof that uses the div_cancel lemma will be incomplete. The following alternative definition does not produce such a TCC solving the problem..

$$div\_cancel2 : \textbf{LEMMA}\ \forall x \in \mathbb{Z} - \{0\}, y \in \mathbb{Z} : \frac{x \times y}{x} = x$$

PVS includes a large collection of conditions which are check against the user supplied specification. And many of these will appear when the specification has flaws.

2. Methodology Description and Goals

The main goal is to provide a methodology to transform term rewrite specification in ELAN to the language of the proof assistant PVS where one can prove correctness and integrity properties. The transferring and work with PVS allows formal verification of the ELAN specification, including checking types, generating and checking the joinability of critical pairs, CPs for short, and checking of other properties and general correctness of the original TRS specification. For the methodology to be practical, the translation has to take in account several aspects of both ELAN and PVS.

The translation process involves several different components and files. There is one key central component responsible for the translation, but other components are required to make it effective and usable. Figure 1 shows these components and how they interact. Files and software are shown as rectangles and boxes with rounded corners, respectively. They have the following functions and contents:

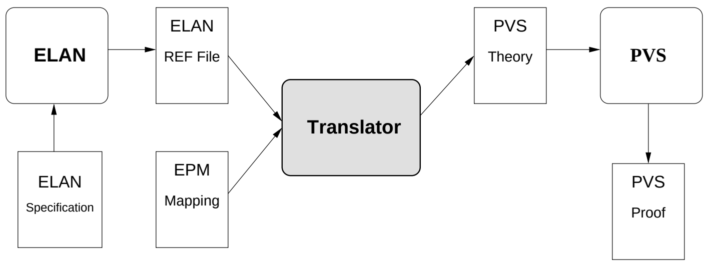

Figure 1: General flow

ELAN Specification: represents the set of files of a given ELAN specification. Most specification will have at least two files: *top level logic description* and *module* files. Some specification will have additional files called *specification* files. All these files are required to use the ELAN interpreter with that specification.

ELAN: This is the ELAN interpreter. It can be used to run specifications. Its role in the translation process is to export the specification into a ELAN REF file.

ELAN REF File: is used by the ELAN Compiler to generate C code for the given specification. This allows ELAN to create executable programs that run a specification. The content is very suitable for computer analysis, and hold all elements necessary for the generation of the specification in other languages. However the specification does not include any variable names (using something similar to De Bruin's notation for $\lambda-$calculus). This file includes many rules that are not visible in the *module* files. These rules come from imported types and modules, as well as rules generated by properties such as associativity and commutativity.

EPM Mapping: Are used to control the translation process and behavior. EPM stands for *ELAN to PVS mapping*. It provides naming for variables, translation for operators and function symbols as well as naming for the rules. ELAN does not have naming for rules, but PVS requires names for declarations such as the ones that will represent the rules. Having meaningful names is important when trying to use rules in proofs. As stated above ELAN will generate internal rules and import rules for modules and types that may be irrelevant to the verification of the specification. To handle this the EPM file can define rules as irrelevant or ignorable.

Translator: is the key component. It will read the ELAN REF file and the EPM mapping files to load the rules into structures that can easily be manipulated. Then, it export the rules to a PVS file, generate the CPs and provide means to check the CPs (without taking in account the feasibility of their conditions). The translation process will be described in greater detail in section 3.

PVS Theory: PVS files contain the theory declarations. Those generated by the translator will have axioms for the rules of the term rewrite specification, lemmas for the CPs, and function and type declarations, all derived from the original specification.

PVS: represents the PVS runtime that can read theory files, do type checking and allow users to prove the theory declaration.

PVS Proof: once the user has proved a theory element or declaration, the proof is saved in a proof file. This allows them to be run in batch to re-validate a theory even if some aspects of the theory have changed.

One may question why the translation does not handle the conditions of the CPs eliminating all CPs with unfeasible conditions, that will trivially proved in PVS. The main reason for not doing so it that PVS has these logical capabilities and introducing decision procedures could produce a very complex translator giving yet another proof engine evaluator which is not the goal in our methodology.

3. Translation from ELAN to PVS

The translation from ELAN specifications to PVS theories is the cornerstone of our methodology. The key idea is that using a semantically intelligent translation, one can emulate the rewriting based specifications and use the proof assistant to check both term rewrite properties as well as non rewrite properties such as logical correctness and integrity. The translation mechanism should respect the semantics of the given input specifications and preserve typing information. PVS and ELAN have a somewhat compatible expression power, but the translation has to deal with differences between them.

Within any ELAN specification there are three key elements: **Types** called *sorts* in ELAN; **Function Symbols** called *operators* that represent operator and function symbols and **Rules** that represent the rewrite rules of TRSs. All these elements are listed in the ELAN REF file and must be transported into the PVS theory. Once all of these are loaded, it is possible to generate CPs, which should be included in the PVS theory.

3.1. Types and Function Definitions

ELAN and PVS are strongly typed system. Both also have parametric types (such as list[int] for list of integers). However PVS has a much more powerful typing system, including dependent and recursive types. A naïve approach is to translate a type from one system to the other directly *name per name*. So an ELAN *sort* of $mytype$ become a PVS type declaration: mytype: **TYPE** .

This approach will generate syntacticly and semantically correct PVS theories, however with complex types such as lists and arrays, this strategy may produce unprovable TCCs. These complex types can be translated into **DATATYPE** types. These type declarations include means to declare constructors and nil elements, and will normally eliminate TCC regarding constructed types.

In ELAN a list can be declared as follows:

```
sort mylist; end
operators global
    mynil: mylist;
    @.@: (int mylist) mylist;
end
```

In the **operators** block each statement defines a function or operator symbol. '@' is a placeholder for a variable of a specific type. The domain of the function is listed within the parenthesis and the range is the name after the parenthesis. So the $mynil$ operator defines a constant of type $mylist$ and '.' defines and operator of type $int \times mylyst \rightarrow mylist$. The type $mylist$ shown above can be translated into the PVS type declaration mylist : **DATATYPE**. Below, the last identifiers after the colon ($mynil?$ and $mylist\_cons?$) are predicates that can be used in other places to determine whether a variable of that **DATATYPE** is constructed by that constructor or not.

```
mylist: DATATYPE BEGIN
    mynil: mynil?
    o(v:int, l:mylist): mylist_cons?
END mylist
```

In ELAN predicates like $mylist\_cons?$ can be expressed with the following rule: [] isMylist_cons(v . l) => **true end**. PVS requires that a predicate be specified for each constructor although names can be generated in many cases. These predicates are used in the construction of recursive functions, and can be used as rule triggers. More elaborate types may require creating the type declarations manually and turn off type translation.

Another important element of ELAN specifications that must be translated is the type description of the operator and function symbols. This should be done carefully because both ELAN and PVS allow different type descriptions for the same symbol.

One of the key challenges in the translation of the ELAN operators comes from the fact that ELAN has a more powerful lexical engine. For this reason some operators cannot be translated directly to PVS operators. For example $r_1 \mapsto Load(r_2)$ can be written in ELAN as r1|−Load(r2) which cannot be translated directly to the language of PVS. To solve this issue, operator symbols that are not supported in PVS must be translated to different symbols. For instance, the ELAN operator @|−Load(@): (int int) instruction can be declared in PVS as Load: [int,int−>instruction]. The general PVS type description is in the form $t_1, t_2, \ldots, t_n \rightarrow i$. Notice that the example above can also be considered a constructor for a **DATATYPE**.

3.2. Rules

ELAN *rules* are translated to PVS *axioms*. This design decision was taken since translating the rules to functional definitions would not necessarily yield adequate functional definitions, and because PVS can emulate TRS rewriting with axioms during proofs.

Within the axiom definition the rewriting operator $\rightarrow$ has to be replaced by the equality operator $=$. Equality in PVS can be used in any direction potentially causing problems. However most proof commands that refer to equalities will do so from left to right by default. This is the expected direction for a TRS. However the use of equality requires special care in the proof to avoid using the equality in the *wrong* direction.

The emulation of a rewrite step within a proof is done by a command called **REWRITE**. This command uses lemmas or axioms and a term or set of terms as arguments. It will attempt to find unifiers between any subterm of terms being rewritten and the left side of the equality in the lemma or axiom. If it can match them, it replaces the subterm in the term by the right side of the axiom or lemma. So $t \xrightarrow{axiom} t'$. This is similar to the semantic of applying a rule in a TRS, $t \xrightarrow{R} t'$. One caveat of the **REWRITE** command is that it can be used from right to left, although the default is left to right.

A unconditional rule written in ELAN as [] a −> b **end**, where the empty braces [] (for unlabeled rules) tells ELAN that this rule is not used by any specific strategy. ELAN can use strategies to control the application of rules. This simple rule is translated to the PVS statement r0: **AXIOM** a=b.

Every axiom and lemma in PVS must have an unique name. These names could be generated automatically, but this is bad for usability since rules have no mnemonic meaning to the user. Another important aspect of naming axioms and lemmas is that proofs and TCCs refer to these names. If a name changes the proofs may be lost, requiring the user to prove them again. So having good names is important to make the user more productive.

ELAN is a conditional rewriting framework. The ELAN statement [] a −> b **if** c **end** means that a will be rewritten to b if the condition c can be evaluated as true and since there is no label (i.e., any strategy identification) it can be used anywhere. The statement above can be translates to the PVS term: r1: **AXIOM** c **IMPLIES** a=b. This imitates the way rules work in ELAN, since the axiom can only be applied if PVS can determine that the condition c holds. Strategies are not dealt by the translation, although this could potentially change the results of the translations.

ELAN and PVS use *where* constructions to simplify rules and make them more readable. These constructions assign values to variables that appear on the right side of the rules. There are however some differences between the two languages. In ELAN the *where* constructions refer to any variable and condition, in PVS the variable used in conditions or on the left side of a axiom or lemma declaration must be defined before they are used in the statement. This is done with a **LET** ... **IN** expression, which has a similar semantics to a regular *where* construction.

The following example illustrates the way a simple *where* construction is translated. Consider the ELAN rule [] a −> c **where** c:=()op(a) **end**. The ELAN assignment operators are in the form :=() or :=[], where the parenthesis or braces can contain the name of a strategy that should be used to evaluate the right side of the assignment. This rule translates to the PVS declaration: r2: **AXIOM** a=c **WHERE** c=op(a). However if the ELAN *where* construction affects a variable in a condition, like in the following rule:
[] a −> b **where** c:=()op(a), **if** c
it must be translated slightly differently. Since the condition has to come before the rule's equality, and the variable of the condition has to be defined in the *where* construction, these must be transformed into the **LET** ... **IN** expression such as:
r3: **AXIOM LET** c=op(a) **IN** c **IMPLIES** a = b

Variables in both languages can be declared prior to their use in the rules. In TRSs right side variables have to occur also in left side of the rules or be determined by left-side variables in a *where* construction. However PVS allows variables to be declared and typed per theory declaration (lemma or axiom). This is done with a **FORALL**$(v_1 : t_1, .., v_n : t_n)$: ... construction (where the v_i's are variables and t_i's their types). This is an alternative to the use of global variables and ensures that translation is done correctly since undeclared variables will cause errors during the type checking of the theory. To illustrate how the variable are declared,assuming a of type int and c is a $boolean$ variables and b is a constant declared previously, the previous rule would actually be translated as:
r4: **AXIOM FORALL**(a:int): **LET** c:boolean=op(a) **IN** c **IMPLIES** a = b.

Additionally, the ELAN rule construction block **choose try** ... **end** should be handled. Theses blocks allow different assignment to variables depending on conditions provided within the block. Each group started by a **try** represents a set of variable assignments and condition that must be meet. If the condition hold, that set of assignments is accepted and no additional blocks are evaluated. Observe the following block.

```
[] f(x) −> g(y,z) choose try where y:=()h(1,x) where z:=()g(y,x) if x < 0
                         try where y:=()h(x,1) where z:=()g(x,y) if x > 0
                         try where y:=()0 where z:=()1 end
```

PVS provides the **COND** or *condition* expression that has a very similar semantics to that of the ELAN **choose try** ... **end** blocks. However each expression returns a single value of a specific type, and is used to assign a value to a single variable. In ELAN, several variables can be affected by the same condition expression. To handle this, translation must de-multiplex the variables into different PVS **COND** expressions. Assuming all the variables are of type *int*, the rule shown above can be translated to the following PVS declaration.

```
rcond: AXIOM FORALL(x:int): f(x)=g(y,z)
      WHERE
        y=COND x<0 −> h(1,x), x>0 −> h(x,1), ELSE −> 0 ENDCOND,
z=COND x<0 ->g(y,x), x>0 ->g(x,y), ELSE -> 1 ENDCOND
```

Each condition in the **COND** expression is expressed in the form $c_i \rightarrow v_i$, which means that if c_i holds v_i is the value of the expression. The condition ELSE $\rightarrow v_{else}$ is used if none of the previous conditions is true. Notice that in this example the conditions are the same in both **COND** expressions, just the values returned change. This translation allows PVS to emulate the general semantics of the ELAN **choose try** ... **end** blocks.

The options taken in the translation of rules were carefully analyzed and tested to ensure the preservation of the TRS semantics as well as the usability of the PVS generated theories. Although the translation can produce quite large PVS declarations, a user familiar with the ELAN specifications can easily identify which rule he is observing and determine if it translation matches the original rule. This knowledge is essential during verification, since users will be choosing which axioms (i.e. ELAN rule) to apply as they issued proof commands.

ELAN strategies were not translated to the PVS theory. In ELAN they are used to suggest to the interpreter ways in which the rules should be used. If rules where applied in a non deterministic form, recursion and other looping structures could cause the large consumption of resources that could render computing a result unfeasible. The strategies where ignored (for the time being) because it is assumed that an intelligent user will be making the decision on which rules to apply and this will avoid the problems of random rule selection.

3.3. Critical Pairs

Once all the mandatory elements of an ELAN specification have been handled, it is possible to generate CPs. These have no ELAN equivalent, though one could attempt to see how ELAN would handle all of them as queries to the ELAN engine. However this approach is not very practical for testing the joinability.

CPs are generated by overlapping left-hand sides of rules and then applying one rule to get the first element of the pair and the other rule to obtain its second. So lets say the following rules have left-sides such that one non variable subterm of the one unifies the other:

$$R_1 : l_1 \rightarrow r_1 \quad \text{and} \quad R_2 : l_2 \rightarrow r_2$$

Assuming this unification produces the term u and that:

$$s \xleftarrow{R_1} u \xrightarrow{R_2} t \quad \text{so the pair is } \langle s, t \rangle$$

CPs are generated to determine whether they are joinable or not, in order to apply rewriting theory to deduce (local-)confluence of the whole TRS. CPs are translated to the PVS theory as conditional equational *lemmas* that need to be proved. The joinability of CPs is determined by proving these lemmas. So for the critical pair $\langle s, t \rangle$ one has to prove by rewriting that s and t can be made equal. The corresponding PVS declaration is the lemma: cp1: **LEMMA** s = t.

Many of the rules that create CPs will have conditions associated to them, and will likely have *where* constructions. Both of them must be considered when creating lemmas corresponding to the CPs. These constructions can be joined, assuming they are properly instantiated for variable renaming and unification.

To illustrate this lets take a somewhat complex set of rules:

```
[] f(x,y) => f(v,z) where v:=()g(x,y) where z:=()g(y,x) if x<y end
[] f(x,h(y)) => g(h(z),h(z)) where z:=()x+y if x>=h(z) end
```

The left-hand side of both rules can be unified to $f(x, h(y))$ (once renaming the variables in the first rule). Applying the first rule the following term will appear:

```
f(v,u) where v:=()g(x,h(y)) where u:=()g(h(y),x) if x<h(y)
```

And applying the second rule the following term will appear:

```
g(h(z),h(z)) where z:=()x+y if x>=h(z)
```

So the resulting PVS lemma must have the pair's terms and all their condition. The general consideration used for rules must be followed in the CPS. Conditions for rule usage, must be translated into **IMPLIES**. The critical pair from the example rules above should be expressed in PVS as follows:

```
1  cp2: LEMMA FORALL(x,y:int): LET z=x+y IN x<h(y) AND x>=h(z) IMPLIES
2          f(v,u) = g(h(z),h(z))
3          WHERE u=g(h(y),x), v=g(x,h(y))
```

The equality in line 2 is the key test for joinability. The *where* construction from the second rule has to be placed before the main equality because it is needed in the **IMPLIES** clause (first line). Line 3 is the *where* construction from the first rule.

On a properly constructed rewrite based specification most of the CPs will have inconsistent conditions, eliminating the CPs altogether. This is a consequence of the fact that the specifier tries to define rewrite rules (for defining each function involved in the specification) whose conditions are exclusive. However, in verifying CPs, even these trivially dispensable, it can appear several problems within the specification as will be discussed in the next session.

4. Practical experiences

This section explores some of the practical experiences in using the methodology and the translator. It explores the verification of a simple processor, a basic non speculative processor, from the work described in [Ayala-Rincón et al., 2002]. During this verification several lessons on translation strategies and actual errors where found in the original specification.

4.1. Validating an AX Basic Processor Specification

Before introducing the experiences it is important to describe the AX processor. It is a simple RISC processor with 7 instructions: $Load$ loads a register with a value from the memory, $Store$ saves register contents to memory, Op adds two register, OpE compares two registers and store the result on another register, Jz branches the program if a register is zero, $Loadc$ loads a register with a constant value, and $Loadpc$ saves the program counter to a register. The *System* has a random access *memory* (implemented as a list of cells) and a *Processor*. The processor is defined as a *program counter*, a *program memory* (which is read only), and a *register file* (list of registers).

During this verification several interesting aspects of the implementation where revealed. These points came from proving both type checking constraints (TCC) and the critical pair joinability. Several of the CPs where unjoinable, one of them revealed a bug in the implementation. All of the issues had limited impact on the actual use of the specification because ELAN evaluates rules in order. However they could cause problems if the order were to be changed or on actual hardware. Some of the difficulties in proving lemmas and obligations derived from options in the translation. Different strategies produced better results.

4.1.1. Bug: Bad Operation Definition

One of the first interesting problems found in the specification had to do with overlapping comparison operations. The $valueofOpE(m, r_1, r_2)$ is used by the OpE instruction and is defined as: r_1 and r_2 are registers to be compared, and m is the type of comparison to be made. When $m = 3,\ (\leq)$ the specification had a flaw, as show by the rules below.

```
[] valueofOpE(3,x,y) => 1 if x <= y end
[] valueofOpE(3,x,y) => 0 if x >= y end
```

These rules can be overlapped producing a critical pair, which is unprovable because $x = y \implies 1 = 0$ which is clearly an error. The critical pair generated by the translator is shown below (in PVS).

```
OpE3_lteXOpE3_gte: FORALL(x:int,y:int): x >= y AND x <=y IMPLIES 1=0
```

This error does not produce problem when using the specification in ELAN because the rules are applied in order. However if the order where to be changed the specification would produce erroneous results since when $x = y$ both $valueofOpE(3, x, y) \rightarrow 0$ and $valueofOpE(3, x, y) \rightarrow 1$ are valid reductions.

4.1.2. Register File Critical Pairs

In the AX implementation register files are implemented as lists of registers. The register is constructed by a function $Reg(i, v)$ where i is the index number of the register and v is it's value. The rules for the update of the register file are shown below.

```
1    [] insertRF(nilr,r,v) => Reg(r,v).nilr end
2    [] insertRF(Reg(i,j).rf,r,v) => Reg(r,v).rf if r == i end
3    [] insertRF(Reg(i,j).rf,r,v) => Reg(i,j).insertRF(rf,r,v) end
```

The first rule is used to add a register to an empty register file or to the end of the file. The second rule is used to replace a register value in the register file, while the third rule advances through the register file. When CPs are analyzed the following critical pair appears from overlapping the second and third rules.

```
irf_foundXirf_adv: LEMMA FORALL(v:int,r:int,rf0:RF,j:int,i:int):
  r=i IMPLIES (Reg(i,j) o insertRF(rf0,r,v))=(Reg(r,v) o rf0)
```

The critical pair has a satisfiable condition and can not be trivially joined. Adding a condition of $r \neq i$ to the third rule changes the conditions of the critical pair turning them unsatisfiable and allowing this way the elimination of the CP. Again the ELAN specification works properly because the rules are executed in order. However, if the second and third rules are swapped the register file specification does not work at all. Adding the condition may seem redundant, but makes the specification clearer and ensures that order will not affect the result of the system.

4.1.3. Instructions Critical Pairs

In the construction of the AX processors a set of operators construct the instructions for the processor. Instructions are constructed using operators like $Jz(r, jaddr)$ and $Load(r, addr)$. The main processing rule has the following format, where Instruction represents one of the AX instructions (for example *isinstJz*):

```
[Instruction] Sys(m, Proc(ia, rf, prog)) =>
    Sys(m, Proc(ia+1, insertRF(rf,r,v), prog))
```

```
where instIa :=() selectinst(prog,ia)
if isinst<Instruction>(instIa)
...
```

The key condition for selecting one of these rules is the $isinst\langle Instruction\rangle$ (like $isinstJz$, and so on). Each of these functions takes an instruction to a boolean and indicates if the instruction is constructed by a $Load$, Jz and so on. When two of these rules are checked for CPs, their left-hand sides are identical and will produce a critical pair. There are however conditions on each of these CPs which comes down to something like:

```
... isinstJz(instIa) AND isinstLoad(instIa) IMPLIES ....
```

It is obvious that since different constructor create each instruction, no instruction can be both a Jz and $Load$ at the same time (or any two instruction constructor for that matter). However PVS does not have that information and the CPs cannot be dismissed or proven joinable. Since $isinst\langle Instruction\rangle$ is not reducible in this form there is no way to get around this problem without some changes in the translation or PVS file.

Several approaches were used to solve this issue. The first approach involve telling PVS that if a instruction is of one type it cannot be of other types. This is done by introducing axioms like:

```
exJz: FORALL(instIa:instruction): isinstJz(instIa) IMPLIES
        NOT isinstLoad(instIa) AND NOT ....
```

These inform PVS that the constructors are different and allow the CPs to be proven unfeasible due to false conditions.

Another alternative involves changing the $isinst\langle Instruction\rangle$ to an expandable construction such as the following:

```
[] decode(Load(r,addr)) => 2 end
[] decode(Jz(r,jaddr)) => 3 end
[] isinstLoad(instIa) => decode(instIa) == 2 end
[] isinstJz(instIa) => decode(instIa) == 3 end
```

This construction means that the critical condition:

```
... isinstJz(instIa) AND isinstLoad(instIa) IMPLIES ....
```

can be rewritten to the following:

```
... decode(instIa) = 2 AND decode(instIa) = 3 IMPLIES ....
```

which PVS can easily assert as being false thus discarding the critical pair altogether.

However the most elegant solution to this problem comes from simply changing the type declaration for $instruction$ from a simple type to a constructed **DATATYPE**. In addition to this, the predicate for each constructor must match those used in the condition of the rules, that is the $isinst\langle Instruction\rangle$. Thus the $instruction$ type in constructed in the following manner (shown in PVS):

```
instruction: DATATYPE BEGIN
    ...
    Load(r:int, addr:int) : isinstLoad
    Jz(r:int, jaddr:int) : isinstJz
    ...
END instruction
```

This construction yields the best results, since PVS can determine that only one constructor can be used at each time. This makes the CPs formed from the instruction processing rules all trivially unsatisfiable.

4.1.4. Jz Condition Clause TCC

The rule for the Jz instruction provided another proof challenge. In it's original construction (partially shown below) it has a **choose try** .. **end** block. As stated in the translation (section 3) this can be converted to a **COND** block. However in doing so this rule and any critical pair that uses it produces a TCC.

```
choose
    try where nia:=()ia+1 if valueofReg(r1,rf) != 0
    try where nia:=()valueofReg(r2,rf) if valueofReg(r1,rf) == 0
```

The reason for this is that PVS requires that all conditions in a **COND** block be pairwise disjoint. Since this block has no *else* condition, proving this TCC requires that $valueofReg(r_1, rf) = 0$ and $valueofReg(r_1, rf) \neq 0$ cannot hold at the same time. This is trivially true, but PVS cannot determine this immediately when type checking. One way to solve this is to eliminate the **if** expression from the second **try** block. This forces the translation to use a **ELSE** condition in the **COND** block which eliminates the TCC.

5. Conclusions and Future Work

The proposed methodology is easily applicable for general term rewriting based specifications and generating the target lemmas corresponding to (joinability of) CPs and using the power of a proof assistant such as PVS simplifies the verification of correctness. And the most important, it allows for dynamic and quick execution of changes in the specification without losing advances in proving its correctness: in PVS proved lemmas not involved in these changes can be reused in new proofs. PVS proved to be very suitable for this task since it can emulate the rewriting semantics involved in conditional rewriting rules and conditional critical pairs very well.

The experience with the basic AX specification showed various important options concerning the translation process. The initial naïve approach produced many unprovable TCCs and had some critical pairs that could not be discarded. With different translation strategies and minor adjustments to the specification, it was possible to create a PVS theory with no TCCs and whose CPs where all discarded. Even with all the care taken in the translation the fact that PVS does not have an order when applying lemmas and axioms, and that types can be handled in different forms can play an important role in the number of TCCs and the complexity of proving properties.

Larger specifications are being converted and tested after the initial success of the methodology. Future work includes creating proof strategies to assist proving the CPs between PVS, and making the translation aware of the ELAN strategies.

Finally, we believe the proposed methodology is a step forward in the direction of describing a verification mechanism for innovative new hardware technologies such as reconfigurable systems. Current work in this setting point out the translation of rewriting based specifications of algebraic operations to alternative equivalent expressions via different logical strategies in ELAN, which are then translated to the language of the ALE-X compiler of the Xputer reconfigurable architecture and subsequently mapped into the PACT XPP commercial architectures [Morra et al., 2005].

References

Arvind and Shen, X. (1999). Using term rewriting systems to design and verify processors. *IEEE Micro Special Issue on "Modeling and Validation of Microprocessors"*, 19(3):36–46.

Ayala-Rincón, M., Nogueira, R. B., Jacobi, R. P., Llanos, C., and Hartenstein, R. (2003). Modeling a reconfigurable system for computing the FFT in place via rewriting-logic. In *16th Symp. on Integrated Circuits and System Design*, pages 205–210. IEEE CS.

Ayala-Rincón, M., Jacobi, R. P., Carvalho, L. G. A., Llanos, C., and Hartenstein, R. (2004). Modeling and Prototyping Dynamically Reconfigurable Systems for Efficient Computation of Dynamic Programming Methods. In *17th Symp. on Integrated Circuits and System Design*, pages 248–253. ACM Press.

Ayala-Rincón, M., Neto, R. M., Jacobi, R. P., Llanos, C., and Hartenstein, R. (2002). Applying ELAN strategies in simulation processors over simple architectures. In *2nd Int.Workshop on Reduction Strategies in Rewriting And Programming*, volume 70(6) of *ENTCS*, pages 127–141. Elsevier.

Becker, J. and Hartenstein, R. W. (2003). Configware and morphware going mainstream. *Journal of Systems Architecture*, 49:127–142.

Brader, F. and Nipkow, T. (1998). *Term Rewriting and All That*. Cambridge UP.

Cyrluk, D., Rajan, S., Shankar, N., , and Srivas, M. (1994). Effective theorem proving for hardware verification. In *Theorem Provers in Circuit Design (TPCD '94)*, volume 901 of *LNCS*, pages 203–222. Springer.

Hoe, J. C. and Arvind (2004). Open-Centric Hardware Description and Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(9):1277–1288.

Kapur, D. and Subramaniam, M. (1997). Mechanizing Verification of Arithmetic Circuits: SRT Division. In *Proc. 17th FSTTCS*, volume 1346 of *LNCS*, pages 103–122. Springer.

Kapur, D. and Subramaniam, M. (2000). Using an Induction Prover for Verifying Arithmetic Circuits. *J. of Software Tools for Technology Transfer*, 3(1):32–65.

Martí-Oliet, N. and Meseguer, J. (2002). Special Issue on Rewriting Logic and its Applications. *TCS*, 285(2).

Miller, S. P. and Srivas, M. (1995). Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, pages 2–16. IEEE CS.

Morra, C., Becker, J., Ayala-Rincón, M., and Hartenstein, R. (2005). FELIX: Using Rewriting-Logic for Generating Functionally Equivalent Implementations. Technical report, Universität Karlsruhe and Universidade de Brasília. Submitted, available under solicitation: `ayala@mat.unb.br`.

Ohlebusch, E. (2002). *Advanced Topics in Term Rewriting*. Springer.

Owre, S., Rushby, J., Shankar, N., and Stringer-Calvert, D. (1998). PVS: An Experience Report. In *Applied Formal Methods-FM-Trends 98*, volume 1641 of *LNCS*, pages 338–345. Springer.

Owre, S., Rushby, J. M., Shankar, N., and Srivas, M. K. (1994). A tutorial on using PVS for hardware verification. In *Theorem Provers in Circuit Design (TPCD '94)*, volume 901 of *LNCS*, pages 258–279. Springer.

R. Hosabettu and G. Gopalakrishnan and M. Srivas (2003). Formal verification of a complex pipelined processor. *Formal Methods in System Design*, 23(2):171–213.

Rushby, J., Owre, S., and Shankar, N. (1998). Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720.

van Rijsbergen, C. J., editor (2003). *Term Rewriting Systems*. Cambridge UP.

A. A Simple Example

We would like to remark that this appendix is included for the benefit of the review process and that it will be dropped in a final version of the paper.

This examples involves a simple ELAN specification with few rules, and then an exercise of the translation and working with the specification. The PVS theory resulting from the translation includes four CPs and is complemented with some lemmas that should be proved to obtain the correctness of the specification. This provides the reader with an example of the potential benefits of using this methodology to verify the ELAN specifications.

The ELAN specification used in this example provides two sets of rules:

- Rules affecting $agp(s, n, i)$ function, creates a list of elements where each element is the $i-$th elements of an arithmetic progression. The list will have n elements, with initial value of i and a increment of s between elements. The rules are shown below (with the actual line number for each rules).

```
21    [] apg(s,n,i) => nililist if n<=0 end
22    [] apg(s,n,i) => i.apg(s, nc,ni) where nc:=() n−1 where ni:=()i+s if n>0 end
```

- Rules affecting $listsum(l)$ and $listsum\_r(l, s)$ that take a list l of elements and adds them up, providing the sum s of all the values in the list.

```
29    [] listsum(nililist) => 0 end
30    [] listsum(l) => listsum_r(l,0) end
31    [] listsum_r(nililist,s) => s end
32    [] listsum_r(h.l,s) => listsum_r(l,ns) where ns:=() h+s end
```

The ELAN specification also define the list of integer with the type $ilist$, which is constructed using the ◦ operator and the $nililist$ for an empty list. The agp rules are both conditional, insuring that the list can be assembled even with $n < 0$.

Once the basic specification has been tested using ELAN, the translation process is done. Beside the translated elements the PVS theory includes several lemmas design to test the correctness of the specification.

- The listing includes the conversion of the rules to PVS. Notice that the "@.@" operator had to be replaced by "@ ◦ @". Besides that the rules in PVS follow much the same appearance of the ELAN rules.

```
15    apg_end: AXIOM FORALL(i:int,n:int,s:int): n<=0
16            IMPLIES apg(s,n,i)=nililist
17    apg_cons: AXIOM FORALL(i:int,n:int,s:int): n>0
18            IMPLIES apg(s,n,i)=i o apg(s,nc,ni)
19            WHERE nc:int=n−1, ni:int=i+s
20
21    listsum_nil: AXIOM listsum(nililist)=0
22    listsum: AXIOM FORALL(l:ilist): listsum(l)=listsum_r(l,0)
23    listsum_r_end: AXIOM FORALL(h:int): listsum_r(nililist,h)=h
24    listsum_r_adv: AXIOM FORALL(s:int,l:ilist,h:int):
25            listsum_r(h o l,s)=listsum_r(l,ns) WHERE ns:int=h+s
```

- During the translation four CPs were found. Notice that the names where given using the rules names, and that two of them are just the reverse evaluation of the rules.

```
28    apg_endXapg_cons: LEMMA FORALL(n:int,s:int,i:int):
29            n<=0 AND n>0 IMPLIES
30            LET nc0:int=n−1,ni1:int=i+s IN i o apg(s,nc0,ni1)=nililist
31    apg_consXapg_end: LEMMA FORALL(n:int,s:int,i:int):
```

```
          n>−1 AND n<=0 IMPLIES
          nililist=i o apg(s,nc0,ni1)
            WHERE nc0:int=n−1, ni1:int=i+s
    listsum_nilXlistsum: LEMMA listsum_r(nililist,0)=0
    listsumXlistsum_nil: LEMMA 0=listsum_r(nililist,0)
```

All these critical pair have unsatisfiable conditions and can be trivially discarded.

In addition to the elements form the ELAN specification, the PVS definitions include three lemmas and a function definition. These declarations are used to prove the operation correctness of the specification as follows:

1. $ilist\_length(l)$ is a recursive function that determines the size of a $ilist$. This is a pure PVS definition, as shown below.

```
    ilist_length(l:ilist): RECURSIVE int = CASES l OF
          nililist : 0,
          o(h, t) : 1 + ilist_length(t)
    ENDCASES MEASURE l BY <<
```

   The **MEASURE** ... **BY** is required by PVS to indicate that this recursion is well defined.

2. The apg_len lemma states that a list generated by the apg rules with argument n has length of n. This is a very simple lemma and seems fairly obvious, never the less it's an example of proving that the rules really do what was intended. It's defined in PVS as follows:

```
    apg_len: LEMMA FORALL(n:nat,i,s:int): ilist_length(apg(s,n,i))=n
```

3. The $listsum\_val\_ext$ lemma is used in the proof of the next lemma. It was required because the way the $listsum$ is defined in the rules, storing the sum s within the functional symbol. (See line 32 of the ELAN specification).

```
    listsum_val_ext: LEMMA FORALL(v,c:int, l:ilist):
          listsum_r(v o l,c)=listsum(l)+v+c
```

4. The last lemma $listsum\_val$ shows that the $listsum(agp(s, n, i))$ with s the increment, i the initial value and n has the values of the closed form of the sum of elements in an arithmetic progression (where a_i is the $i-$th element of the arithmetic progression:

$$listsum(agp(s,n,i)) = \frac{(a_0 + a_n) \times n}{2} \quad \text{where } a_0 = i \text{ and } a_n = i + (n-1) \times s$$
$$= \frac{(i + (i + (n-1) \times s)) \times n}{2} = \frac{(2i + (n-1) \times s) \times n}{2}$$

   In PVS this is expressed as follows:

```
    listsum_val: LEMMA FORALL(n:nat,i,s:int):
          listsum(apg(s,n,i))= ((2 ∗ i + s ∗ l −s)∗l)/2
          WHERE l:int=ilist_length(apg(s,n,i))
```

The proof of the first lemma can be done with a simple induction on n. However the ELAN specification specifies n as $n \in \mathbb{Z}$, which is not suitable for an induction. The solution is to change n to a $n \in \mathbb{N}$ in the definition of the lemma. Since agp is defined for $\mathbb{Z}$ and $\mathbb{N} \in \mathbb{Z}$, the definition will work properly.

The next lemma, $listsum\_val\_ext$, is also proved by an induction, but this time it is done on the l (an $ilist$). This is an interesting feature of PVS, where any well-founded type can be used for an induction. This lemma is required for the next lemma since the sum of the list is store within the $listsum\_r(l, s)$ function as the s value. However for the next induction it is important to have the value of the accumulated sum outside of the function symbols. This is what this lemma proves, that on each step of the $listsum$:

$$\forall v, c \in \mathbb{Z}, l \in ilist : listsum\_r(v \circ l, c) = listsum(l) + c + v$$

The last lemma $listsum\_val$, is proved by induction using the previous lemmas to assist in the proof. Again in order to have an induction it is necessary to restrict n to $n \in \mathbb{N}$.

As shown above using PVS we can check more than the basic properties of a rewrite logic specification. One other interesting aspect of using PVS is that the ELAN specifications goes through a comprehensive type checking. In this simple example there are no type checking constrains with the basic ELAN specification. However in more complex specification this checking helps users detect potential problems.

Listing 1: ELAN specification simple.eln

```
module simple
import global int;
end

sort
        ilist;
end

operators global
        apg(@,@,@) : (int int int) ilist;
        listsum(@) : (ilist) int;
        listsum_r(@,@) : (ilist int) int;
        @.@ : (int ilist) ilist;
        nililist  : ilist;
end

rules for ilist
        s,n,i,nc,ni: int;
   global
        // apg ( step, count, increment)
        [] apg(s,n,i) => nililist if n<=0 end
        [] apg(s,n,i) => i.apg(s, nc,ni) where nc:=() n−1 where ni:=()i+s if n>0 end
end

rules for int
        l:ilist;
        h,s,ns:int;
        global
        [] listsum(nililist) => 0 end
        [] listsum(l) => listsum_r(l,0) end
        [] listsum_r(nililist,s) => s end
        [] listsum_r(h.l,s) => listsum_r(l,ns) where ns:=() h+s end
end
end
```

Listing 2: PVS theory simple.pvs

```
simple: THEORY
```

```
BEGIN

    % ilist constructor
    ilist: DATATYPE BEGIN
    o(v:int,l:ilist): ilist_oh?
    nililist: nililist?
    END ilist

    apg : [int,int,int->ilist]
    listsum : [ilist->int]
    listsum_r : [ilist,int->int]

    % Rules
    apg_end: AXIOM FORALL(i:int,n:int,s:int): n<=0
        IMPLIES apg(s,n,i)=nililist
    apg_cons: AXIOM FORALL(i:int,n:int,s:int): n>0
        IMPLIES apg(s,n,i)=i o apg(s,nc,ni)
        WHERE nc:int=n-1, ni:int=i+s

    listsum_nil: AXIOM listsum(nililist)=0
    listsum: AXIOM FORALL(l:ilist): listsum(l)=listsum_r(l,0)
    listsum_r_end: AXIOM FORALL(h:int): listsum_r(nililist,h)=h
    listsum_r_adv: AXIOM FORALL(s:int,l:ilist,h:int):
        listsum_r(h o l,s)=listsum_r(l,ns) WHERE ns:int=h+s

    % Critical Pairs
    apg_endXapg_cons: LEMMA FORALL(n:int,s:int,i:int):
        n<=0 AND n>0 IMPLIES
        LET nc0:int=n-1,ni1:int=i+s IN i o apg(s,nc0,ni1)=nililist
    apg_consXapg_end: LEMMA FORALL(n:int,s:int,i:int):
        n>-1 AND n<=0 IMPLIES
        nililist=i o apg(s,nc0,ni1)
          WHERE nc0:int=n-1, ni1:int=i+s
    listsum_nilXlistsum: LEMMA listsum_r(nililist,0)=0
    listsumXlistsum_nil: LEMMA 0=listsum_r(nililist,0)

    %PVS Elements used to prove
    ilist_length(l:ilist): RECURSIVE int = CASES l OF
        nililist : 0,
        o(h, t) : 1 + ilist_length(t)
    ENDCASES MEASURE l BY <<


    apg_len: LEMMA FORALL(n:nat,i,s:int): ilist_length(apg(s,n,i))=n

    listsum_val_ext: LEMMA FORALL(v,c:int, l:ilist):
        listsum_r(v o l,c)=listsum(l)+v+c

    % Sum is of pg is ((e_0 + e_n)*n)/2
    listsum_val: LEMMA FORALL(n:nat,i,s:int):
        listsum(apg(s,n,i))= ((2 * i + s * l -s)*l)/2
        WHERE l:int=ilist_length(apg(s,n,i))

END simple
```