COMPUTATION TOOLS 2011

The Second International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking

ISBN: 978-1-61208-159-5

September 25-30, 2011
Rome, Italy

COMPUTATION TOOLS 2011 Editors

Kenneth Scerri, University of Malta, Malta

Pascal Lorenz, University of Haute Alsace, France

COMPUTATION TOOLS 2011

Foreword

The Second International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking [COMPUTATION TOOLS 2011], held between September 25 and 30, 2011 in Rome, Italy, continued an event under the umbrella of ComputationWorld 2011 dealing with logics, algebras, advanced computation techniques, specialized programming languages, and tools for distributed computation. Mainly, the event targeted those aspects supporting context-oriented systems, adaptive systems, service computing, patterns and content-oriented features, temporal and ubiquitous aspects, and many facets of computational benchmarking.

We take here the opportunity to warmly thank all the members of the COMPUTATION TOOLS 2011 Technical Program Committee, as well as the numerous reviewers. The creation of such a broad and high quality conference program would not have been possible without their involvement. We also kindly thank all the authors who dedicated much of their time and efforts to contribute to COMPUTATION TOOLS 2011. We truly believe that, thanks to all these efforts, the final conference program consisted of top quality contributions.

Also, this event could not have been a reality without the support of many individuals, organizations, and sponsors. We are grateful to the members of the COMPUTATION TOOLS 2011 organizing committee for their help in handling the logistics and for their work to make this professional meeting a success.

We hope that COMPUTATION TOOLS 2011 was a successful international forum for the exchange of ideas and results between academia and industry and for the promotion of progress in the areas of computational logics, algebras, programming, tools, and benchmarking.

We are convinced that the participants found the event useful and communications very open. We also hope the attendees enjoyed the charm of Rome, Italy.

COMPUTATION TOOLS 2011 Chairs:

Zhiming Liu, UNU-IIST, Macao
Jaime Lloret Mauri, Polytechnic University of Valencia, Spain
Radu-Emil Precup, "Politehnica" University of Timisoara, Romania
Kenneth Scerri, University of Malta, Malta
Torsten Ullrich, Fraunhofer Austria Research GmbH - Graz, Austria

COMPUTATION TOOLS 2011

Committee

Table of Contents

Netty: A Prover's Assistant

Eric Hehner and Lev Naiman
Department of Computer Science
University of Toronto
40 St. George Street, Toronto, Canada
Email: naiman@cs.toronto.edu

***Abstract*—Netty is a prover's assistant. It supports a calculational style of proof, and allows the creation of proofs that are correct by construction. It provides an intuitive interface that allows direct manipulation of expressions. The key idea is that instead of tactics for proof the user only needs to select the next step of the proof from a list of suggestions. These suggestions are usually the result of unification. Netty provides mechanisms to facilitate proving, such as making use of local assumptions and the use of monotonicity in subproofs. Program refinement from specification can be done in Netty similarly to any other kind of proof.**

***Index Terms*—proof; calculation; tool**

I. Introduction

Proof assistants have been created with various purposes. One such purpose is complete automation; this is useful when we need a proof and we care about the result rather than the process. Such provers have been successful, although the theories they can reason about are limited. Some add freedom of theory by being only partially automatic, requiring some user guidance to produce a proof, and are known as interactive theorem provers. Examples of such provers include HOL, PVS, and Coq [3][10][4]. Some tools have specific modes for program verification. Other tools are specific to program verification or refinement, and hence are restrictive to the theories allowed [8]. Tools like Isabelle [9] require the user to use tactics for proving, which requires the user to learn another meta language. The most similar existing tool to Netty is KeY [2], since it allows users to pick a rule to apply from a list of applicable rules. In contrast, Netty allows users to pick the result of applying rules. KeY does not support the use of local assumptions or monotonicity, and is restricted to the theories it allows.

Despite their usefulness, these tools present a difficulty when used to teach logic. There are often complicated tactics for proof, and perhaps some meta-language for performing certain operations, or a scripting language for creating user-defined tactics. This means that a user must learn a concept that is almost as complicated as programming before being able to prove any expression, regardless of how simple it is.

Netty [7] is a prover's assistant named for Antonetta van Gasteren (1952-2002), a pioneer of calculational proof. It is based on work by Robert Will [11]. Its main purpose is pedagogical; it aims to foster understanding about how a proof is constructed and why each step is allowed. It supports a calculational type of proof, that is similar to the successfully used Structured Derivations [1]. It allows the direct manipulation of expressions and subexpressions to advance a proof. Advancing a proof is usually done by picking an expression from a list of suggestions that Netty provides to be the next line. The importance of this is that it allows a user to concentrate on the proof itself rather than learning how to use the tool.

The paper is organized as follows: Section II discusses the use and structure of calculational proof. Section III shows how the main parts of Netty are integrated. Section IV shows the structure, display and navigation of proofs in Netty. Section V-E is about advancing a proof; it discusses how Netty generates suggestions to proceed with proofs and how the user interacts with the tool to advance the proof. This section also discusses the special mechanisms in proof, and how suggestions are filtered. Section VI shows how a program refinement is performed (identical to any other proof).

II. Calculational Proof

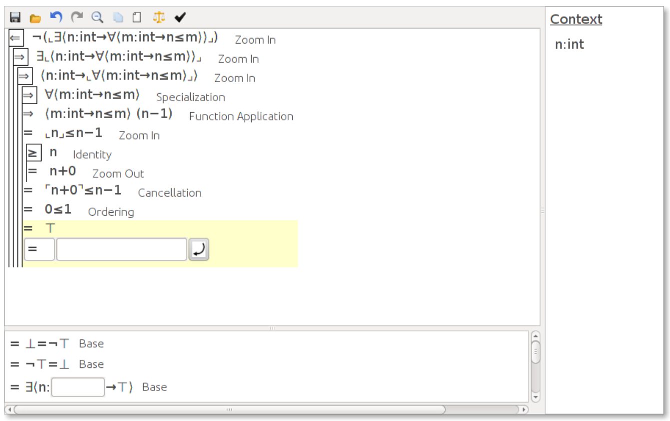
Calculational proof is a fixed and structured format for presenting proofs. It makes proofs and calculations equivalent in that each step has an explicit justification, usually a law. A human prover may have a reason for constructing a proof, and they may have a proof strategy in mind, but these are not our concerns; our concern is to provide a tool that makes proof construction easy and fully formal. The advantage of a fully formal proof is that its correctness is machine checkable. It has been adopted by several researchers in formal methods and used effectively for teaching mathematics at a high-school level [1]. A calculational proof is a bunch of expressions with connective operators between them, whose transitive relation allows us to conclude something about the first and last expression of the proof.

For example, a calculational proof that there is no smallest integer might look like this:

$$\begin{array}{ll}
\neg\exists\langle n : int \to \forall\langle m : int \to n \le m\rangle\rangle & \text{Specialization} \\
\Leftarrow \neg\exists\langle n : int \to \langle m : int \to n \le m\rangle\,(n-1)\rangle & \text{Function Apply} \\
= \neg\exists\langle n : int \to n \le n-1\rangle & \text{Identity Law} \\
= \neg\exists\langle n : int \to n-0 \le n-1\rangle & \text{Cancellation} \\
= \neg\exists\langle n : int \to 0 \le -1\rangle & \text{Ordering} \\
= \neg\exists\langle n : int \to \bot\rangle & \text{Quantifier Identity} \\
= \top &
\end{array}$$

The top line of this proof can be read "there does not exists n of type integer, such that for all m of type integer,

n is less than or equal to m", and the bottom line can be read as "true". The angle brackets serve as the scope of a function and as the scope of a quantified variable (discussed in Section V-C). Notational peculiarities are not the point here; any reader interested in the notational details is referred to [6]. We say that a proof proves an expression '$expn$' if we show that $expn = \top$ or $\top \Rightarrow expn$, and we say that it proves $\neg expn$ if $expn = \bot$ or $expn \Rightarrow \bot$. Hence in this example we say we proved $\neg\exists\langle n : int \rightarrow \forall\langle m : int \rightarrow n \leq m\rangle\rangle$. Each line in the proof has a hint to its right telling how the next line is created. For example, the line $\neg\exists\langle n : int \rightarrow n \leq n-1\rangle$ has the hint 'Identity Law' saying that n is replaced by $n - 0$ in the next line.



```
¬(⌞∃⟨n:int→∀⟨m:int→n≤m⟩⟩⌟)   Zoom In
∃⌞⟨n:int→∀⟨m:int→n≤m⟩⟩⌟   Zoom In
⟨n:int→⌞∀⟨m:int→n≤m⟩⌟⟩   Zoom In
∀⟨m:int→n≤m⟩   Specialization
⇒ ⟨m:int→n≤m⟩ (n−1)   Function Application
= ⌞n⌟≤n−1   Zoom In
n   Identity
= n+0   Zoom Out
= ⌜n+0⌝≤n−1   Cancellation
= 0≤1   Ordering
= ⊤
=

= ⊥=¬⊤   Base
= ¬⊤=⊥   Base
= ∃⟨n:      →⊤⟩   Base

Context
n:int
```

Fig. 2. Calculation Window

III. Overview

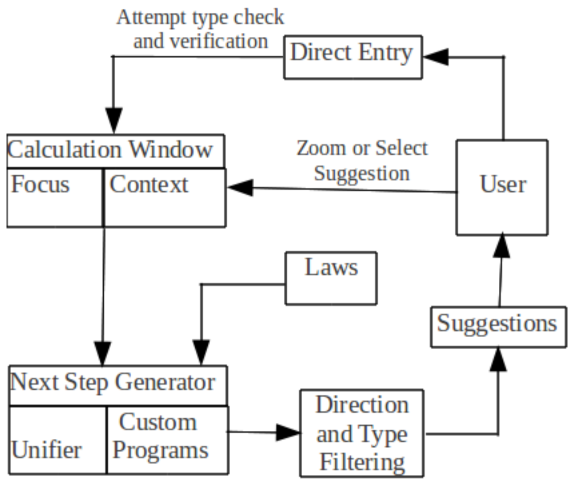


Fig. 1. Backbone

This flowchart illustrates the basic components of Netty. Once a user starts a proof, next steps are generated and filtered as described in Section V. The user can then accept a suggestion, directly enter the next expression, or navigate the proof as described in Section IV.

IV. Proof Display and Navigation

Figure 2 is a screenshot of Netty's calculation window. It is divided into three parts: the proof pane (top left), the suggestions pane (bottom left), and the context pane (right).

The proof pane contains the proof that has been built so far, in the format described in the examples to follow. The suggestion pane contains valid possible next steps in the proof. The context pane displays the laws that are local to the current context. In addition, there is one line selected by the user from which to continue the proof, and it will be referred to as the 'focus'. The box contains the direction in which the proof is allowed to proceed, and hence limits the suggestions that Netty provides for advancing the proof. The proof in Figure 3 serves to explicitly show every step of a Netty proof. Without the boxed directions and vertical lines, we would have a proof that very closely resembles a Structured Derivation that has a main proof and several nested subproofs. The purpose of the vertical lines is structural; they serve to mark the extent of a proof (or subproof). The low corner brackets mark the subexpression that is used in the following subproof, and the high corner brackets mark the result of the subproof.

0.	$\boxed{\Leftarrow}\neg\llcorner\exists\langle n : int \rightarrow \forall\langle m : int \rightarrow n \leq m\rangle\rangle\lrcorner$	Zoom-in
1.	$\boxed{\Rightarrow}\exists\llcorner\langle n : int \rightarrow \forall\langle m : int \rightarrow n \leq m\rangle\rangle\lrcorner$	Zoom-in
2.	$\boxed{\Rightarrow}\langle n : int \rightarrow \llcorner\forall\langle m : int \rightarrow n \leq m\rangle\lrcorner\rangle$	Zoom-in
3.	$\boxed{\Rightarrow}\forall\langle m : int \rightarrow n \leq m\rangle$	Specialization
4.	$\Rightarrow \langle m : int \rightarrow n \leq m\rangle\,(n-1)$	Function Apply
5.	$= \llcorner n\lrcorner \leq n-1$	Zoom-in
6.	$\boxed{\geq}n$	Identity Law
7.	$= n+0$	Zoom-out
8.	$= \ulcorner n+0\urcorner \leq n-1$	Cancellation
9.	$= 0 \leq -1$	Ordering
10.	$= \bot$	Zoom-out
11.	$\Rightarrow \langle n : int \rightarrow \ulcorner\bot\urcorner\rangle$	Zoom-out
12.	$\Rightarrow \exists\ulcorner\langle n : int \rightarrow \bot\rangle\urcorner$	Zoom-out
13.	$\Leftarrow \neg\ulcorner\exists\langle n : int \rightarrow \bot\rangle\urcorner$	Quantifier Identity
14.	$= \top$	

Fig. 3. Detailed Proof

A. Directions and Connectives

The idea of the direction is that in order to conclude some relationship between the first and last line of a proof the connectives for each line must have a transitive relation. For example, the direction $\Leftarrow$ in line 0 allows lines 13 and 14 to use either the $=$ or $\Leftarrow$ connective; the direction $\Rightarrow$ in line 3 allows lines 4, 5, 8, 9, and 10 to use either the $=$ or $\Rightarrow$ connective; the direction $\geq$ in line 6 allows line 7 to use any of the connectives $=$, $\geq$ or $>$. A direction of $=$ allows only the $=$ connective. Notice that the direction symbol must have the

$$\begin{array}{|ll}
\boxed{\Leftarrow} \neg\exists\langle n : int \rightarrow \llcorner\forall\langle m : int \rightarrow n \leq m\rangle\lrcorner\rangle & \text{Specialization} \\
\Leftarrow \neg\exists\langle n : int \rightarrow \ulcorner\llcorner\langle m : int \rightarrow n \leq m\rangle\,(n-1)\lrcorner\urcorner\rangle & \text{Function} \\
 & \text{Application} \\
= \neg\exists\langle n : int \rightarrow \ulcorner\llcorner n\lrcorner \leq n-1\urcorner\rangle & \text{Identity Law} \\
= \neg\exists\langle n : int \rightarrow \ulcorner\llcorner n-0\urcorner \leq n-1\lrcorner\rangle & \text{Cancellation} \\
= \neg\exists\langle n : int \rightarrow \ulcorner\llcorner 0 \leq -1\lrcorner\urcorner\rangle & \text{Ordering} \\
= \neg\exists\langle n : int \rightarrow \ulcorner\bot\urcorner\rangle & \text{Quantifier Identity} \\
= \top &
\end{array}$$

Fig. 4. Compressed Proof

same type as the expression in the proof. In such a way Netty can combine several theories, such as booleans, numbers, and sets.

The direction remains constant throughout a single level of proof, but can change as we zoom into or out of a subproof. In line 0 the direction is $\Leftarrow$ but we zoom past the $\neg$ sign changing the direction in line 1 to $\Rightarrow$.

The line in a parent proof directly after a subproof must also have a connective. Without the subproof it is just a regular line of proof, except that the justification for it is what the subproof proves. It is as if the subproof was a lemma about a subexpression of an expression in the main proof, which is used just like any law, preserving any monotonic properties. This is shown in line 7 of the example in Figure 4. The connective for the subsequent line is determined as follows:

1. If the subproof proves equality, the new leading connective is $=$
2. Otherwise the new leading connective is the direction of the line we zoomed in from

The subsequent line is the same as the line before the subproof, except that the subexpression that was considered in the subproof is replaced with the last line of the subproof.

Such support for monotonicity is not strictly necessary in order to produce proofs. However, it saves many unnecessary steps, and is especially useful for program refinement where a specification is strengthened to a program.

B. Zoom

The suggestions that are created for each line are dependent on the whole expression, mostly because laws involve unification. It would be terribly inefficient and cluttered if we were to produce suggestions that included manipulations on each subexpression. This is why unlike Fitch-style natural deductions [5] we allow subexpression selection. By selecting a subexpression of the focus we say we zoom-in, and hence create a subproof. We can also return one level higher to the parent proof by pushing a key, and we call this zooming out. Only the direct subexpressions of the focus are available for zoom. For example, while doing the proof in Figure 3, on line 5 we have the expression $n \leq n - 1$. The parts that can be zoomed into to create a subproof are n on the left and $n - 1$ on the right.

Zooming is done one level at a time, and getting to a deeper subexpression is simply several zoom actions. The idea is to make selection gestures simple, and not to require a high degree of accuracy in clicking. It might appear useful to be able to select a deeper subexpression directly in order to proceed faster in the proof. However, this is actually not faster. Selecting the right expression takes longer, and navigating to other subexpressions also takes longer. Finally, the presentation layer removes any unnecessary lines, as shown in figure 4.

In Netty expressions are stored in a tree structure with four main classes: literals, variables, scopes, and (function) applications. Literals represent values like 1, $\top$, or 3.14. Variables are not bound to any specific value, but they can be instantiated during unification. Both variables and literals never have any child nodes. Scopes are used to formally introduce variables, give variables types, and to serve as functions. A scope is of the form $\langle var : domain \rightarrow body\rangle$. Applications have an operator (of arbitrary fixity and number of keywords) as the root, and operands as children. The reason that the internal expression structure does not use curried functions for all operators is to allow easy manipulation and use of associativity in the presentation layer. The zoom mechanism then works simply by making a sub-proof initially contain the sub-expression that was selected by the user.

V. Advancing a Proof

A. Unification

The most used algorithm to generate next steps for a proof is unification. The standard unification algorithm is used, with the exception that only the law will have variables that can be unified with sub-expressions of the focus. We differentiate variables and constants by universally quantifying variables. For example, having the law $\forall\langle x : int \rightarrow x \geq 4 \Rightarrow x > pi\rangle$ the unifier would attempt to unify $x \geq 4 \Rightarrow x > pi$ with the focus, treating only x as a variable but pi as a constant. This both provides a form for laws that is fully formal, and allows Netty to distinguish variables from constants. In addition, if an expression matches a law completely, then one of the suggestions given is $\top$. Similarly, if the focus is $\top$ then all laws match. In addition, variables that are introduced in local scopes differ from each other even if they have the same identifier. In the one-point law $\exists\langle v : D \rightarrow v = a \wedge fv\rangle = \langle v : D \rightarrow fv\rangle\ a$ the v on the left side is a different variable than the v on the right side, and can hence be unified with different expressions.

Several law files can be used in Netty; these are plain-text files which are read and parsed by Netty. They can be created, deleted and modified by using any text editor. The laws themselves are simply expressions.

Every line in a proof has some justification at its right hand side (or between lines if more space is needed). This justification is usually the application of some law, and hence it is the name of the law that was applied. In addition, sometimes there are steps that were not justified by a law, or where the type checker could not determine if the focus was of the right type for the law. In these cases, we have a warning symbol, indicating that a certain step is unchecked.

We have the Law Query display mechanism to show a user how the next expression came about (that is, to demonstrate unification). It can also be used to illustrate how the unification algorithm works; for example, suppose somewhere in a proof, we have the two lines:

$$\begin{array}{ll} \quad x \wedge y \Rightarrow (y \vee z \Rightarrow (z \Rightarrow x)) & \text{portation} \\ = x \wedge y \wedge (y \vee z) \Rightarrow (z \Rightarrow x) & \end{array}$$

A click and hold will replace those lines in place by:

$$\begin{array}{ll} \quad a \ \Rightarrow (\ b \ \Rightarrow \ c \ ) & \text{portation} \\ = \ a \ \wedge \ b \ \Rightarrow c & \end{array}$$

The fully formal law is:

$$\forall\langle a, b, c : bool \rightarrow (a \wedge b \Rightarrow c) = (a \Rightarrow (b \Rightarrow c))\rangle$$

The Law Query shows the correspondence (unification) between the variables of the law body and subexpression in the proof.

Occasionally one side of a law matches the focus, and the other side of the law has unconstrained variables. Suppose we have 0 as the focus. Then the right side of the law $x - x = 0$ would match it, and the resulting suggestion would be $= x - x$, leaving x unconstrained [7]. We can have an arbitrary expression placed instead of x, and we cannot generate all possible suggestions for it. Instead, for each unconstrained variable Netty has a dialog box in which a user can type. What is typed in one box for a given variable is inserted into all boxes for that variable, so that the user only has to type it in once.

B. Context

Context is a bunch of local laws in a proof. The idea of context is to be able to make use of local assumptions and to allow the user to only worry about the current expression. At the top-level proof we have no local laws except for the ones the user loaded from law files (discussed in Section V-E). Zooming into subexpressions adds context, and zooming out removes them. This implies that subproofs inherit context from their parent proofs. Context expressions are used exactly like laws, and hence suggestions are generated from them in the exact same manner. The mechanism of context removes the need for any explicit declaration of assumptions, since they can simply be added as an antecedent to the top-level expression. Here are some examples of context rules [6]:

- From $a \wedge b$, if we zoom in on a , we gain context b .
- From $a \vee b$, if we zoom in on a , we gain context $\neg b$.
- From $a \Rightarrow b$, if we zoom in on a , we gain context $\neg b$.
- From $a \Rightarrow b$, if we zoom in on b , we gain context a .
- From **if** a **then** b **else** c **fi** , if we zoom in on a , we gain context $b \neq c$.
- From **if** a **then** b **else** c **fi** , if we zoom in on b , we gain context a .
- From **if** a **then** b **else** c **fi** , if we zoom in on c , we gain context $\neg a$.
- From $\langle var : domain \rightarrow body\rangle$, if we zoom in on $body$, we gain context $var : domain$

To understand the context rules, consider the first one. If we assume b when we zoom into a and b turns out to be true, then we made the right choice. However, if b turns out to be false, there is no harm done in any change to a that assumed b, since the value of the entire expression remains the same (false). Similarly, in the body of a function we can assume $var{:}domain$, since a function must be applied to an element of its domain.

Internally there is a stack of lists that keeps track of context; we add a list of expressions to the context pane on a zoom-in if a context rule is satisfied, and we pop a list on a zoom-out. A zoom-in can add more than one expression to the current context, since t he context expressions are then broken down; each conjunct is gained as a separate law, and if the expression gained is a negation, we push it down the expression tree and perform a deep negation.

C. Scope

Expressions can contain functions, which declare variables. A function has the form $\langle var : domain \rightarrow body\rangle$. A user can zoom into the body similarly to a zoom on any other expression. Any mention of var within the function scope refers to the locally declared var, which is different than any other variable outside the scope with the same name. When we zoom into the scope we might already have in the current context expressions that include var. However, since it is really not the same variable, such expressions cannot be used inside the scope. Netty displays such unusable context expressions at the bottom of the context list in grey. This is to indicate that although we have not lost the context, it cannot be used at present.

The type of variables is gained from their declaration within a scope. If for example we want to prove $\neg a \Rightarrow (\neg b \Rightarrow \neg a)$, we need to start with $\langle a, b : bool \rightarrow \neg a \Rightarrow (\neg b \Rightarrow \neg a)\rangle$, which gives the context of $a : bool$ and $b : bool$ when we zoom into the function body. This way we maintain full formality and give information to the type-checker.

D. Type Checking

The type checker is currently very basic. Literals are given types when the expression is parsed, and variables are given a type if they are declared through a function scope to have a certain type. In addition, the type checker can be invoked with a list of context expressions. In that case the type checker attempts to find the type for any variables whose type has not yet been determined. Functions can have multiple operands and resulting types, which are read in through a configuration file. For example, the type of $\wedge$ might be $bool \rightarrow bool \rightarrow bool$. If both operands are of type $bool$, then the type checker concludes that the type of the expression is $bool$. The reason that functions can have many types is both to allow overloading of functions and to allow greater freedom of theory; this method allows users to define a theory simply through its axioms.

E. Custom Suggestions

The most common way to proceed with a proof is to pick a suggestion by clicking on one from the suggestion pane. Most of the suggestions there are a result of unification with a law. In addition to laws, we use programs to generate suggestions. For example, it would be rather tedious to prove that $5+4 = 9$ using only the construction axiom of natural numbers. Instead we use a program that performs addition and outputs $= 9$ as a suggestion to $5 + 4$. From the user's perspective there is little difference, since the suggestions and justifications appear in the exact same way.

Custom programs can make use of the available context, law and current expression to generate suggestions. For example, an assignment statement such as $x{:=}e$ means that x is assigned e, but every other variable in the state space remains the same. If x and y are state variables, $x{:=}e \;=\; x'{=}e \wedge y'{=}y$. However, in this theory the state space is not explicitly stated in the assignment, but would be available in the context since state variables need to be declared. The custom suggestion generator searches the context and laws for all state variable declarations and outputs a suggestion. In a sense, custom programs are the equivalent of tactics in other provers, since they provide a means other than unification to proceed with a proof.

There is one type of suggestion that is special: function application. If we have $\langle v : d \rightarrow b\rangle\ x$ as the focus, the suggestion given is the result of applying the function to its argument. Function application is not done by unification but by a program, and hence it appears in the same manner as a suggestion for addition. The difference is that in order to apply a function to an argument it must be in the domain of the function. For trivial checks, such as $1 : nat$ or $a : nat$ where we have the type of a in context, we have a simple type checker to perform that check. However, in Netty the concept of type is more general: the type of a variable is just some bunch that the variable is an element or sub-bunch of [6]. This means that with the added power of this theory comes the burden of non-trivial type checking. This is resolved in the Logical Gaps and Direct Entry section.

F. Logical Gaps and Direct Entry

Direct entry is provided through a dialog box below the current line which allows a user to type the next line of the proof. This allows a proof to proceed when a part of it is still unknown or there is no law for it yet. Proofs in Netty are not required to be completely formal at all stages, and steps without formal justification are marked with a warning sign to indicate a possible logical gap in the proof. A user can return to the unsafe line at any time to complete the proof.

Another method of introducing a logical gap is where the type-checker for function application cannot determine if the operands are of the right type. In that case we have a subproof between the previous line and the focus that requires the user to prove the operands are of the right type. For example, if the function application in figure 5 was done somewhere in a proof, the three dots indicate the need to complete a subproof.

$$\boxed{=}\langle n : nat \rightarrow n-1\rangle\ (i \times i)$$
$$\boxed{\Leftarrow} i \times i : nat$$
$$\ldots$$
$$\Leftarrow \top$$
$$= i \times i - 1$$

Fig. 5. Type Proof

This kind of subproof is different in that it does not result from zooming into a subexpression. It can be viewed as a lemma proved in context. A similar gap in the proof would happen if we were to apply a law where the type of the operands is unclear. For the standard types, such as *bool* and *nat* the type checker resolves almost all such problems. It is only for complex types like lists where such a burden of proof is necessary, as it might be non-trivial.

VI. Program Refinement

Program refinement is done in the exact same manner as any other proof using the program theory described in [6]. In the following example the task is to write a program that cubes a number n using only addition, subtraction, and test for zero. We will use two additional state variable x and y. The initial specification is $x' = n^3$.

0. $\boxed{\Leftarrow}\textbf{var}\ x, y, n : nat \cdot \llcorner x' = n^3 \lrcorner$
1. $\boxed{\Leftarrow} x' = n^3$
2. $\Leftarrow x' = n^3 \wedge y' = n^2 \wedge n' = n$
3. $\Leftarrow \textbf{if}\ n = 0\ \textbf{then}\ x := 0.\ y := 0\ \textbf{else}$
   $n := n - 1.\ x' = n^3 \wedge y' = n^2 \wedge n' = n.\ n := n + 1.$
   $\llcorner x' = x + 3y + 3n - 2 \wedge y' = y + 2n - 1 \wedge n' = n \lrcorner\ \textbf{fi}$
4. $\boxed{\Leftarrow} x' = x + 3y + 3n - 2 \wedge y' = y + 2n - 1 \wedge n' = n$
5. $= x := x + y + y + y - n - n - n - 2.$
   $y := y + n + n - 1$

Fig. 6. Refinement

In Figure 6, we see an example of steps that are not fully formal. Netty allows hiding lines, which might be used when certain lines are deemed obvious in the proof. Lines 3 and 5 are the results of either direct entry, line hiding, or a combination of the two. If any line in the hidden or directly entered lines is unsafe, the resulting line will have a warning symbol as a justification. Allowing unsafe lines allows the user to take larger steps in refinement and then return to fill in the gaps. The resulting program without the proof, which we call *cube*, is:

```
procedure $cube$ is
  if $x = 0$ then $x := 0.\, y := 0$
  else $n := n - 1.\, cube.\, n := n + 1.$
      $x := x + y + y + y - n - n - n - 2.$
      $y := y + n + n - 1$
  fi
```

As in the first example, the only necessary actions are a combination of applying laws to change the focus and zooming in on a subexpression to refine. Once all subexpressions have been refined to a program the user will have refined the entire program. The benefit that the zoom mechanism provides is that it allows a user to safely ignore any other subexpression, while having full use of all the local laws in the current context. For example, in order to perform assignments Netty must know all of the state variables. This information is obtained simply by examining the context laws; if we have in context $x : nat$ and $x' : nat$, Netty will conclude that x is a state variable of type nat.

VII. Conclusion and Future Work

Netty has been designed to make proving in the calculational style easy, and we incorporate programming by refinement smoothly as a special case. A lot of attention has been paid to the user interface and ease of use. All methods of proceeding with a proof have been delegated to the generation of next step suggestions, while providing the user with a convenient method of utilizing local assumptions and monotonicity.

Netty has been implemented in Java, using standard GUI libraries such as swing. We have not yet done any empirical studies to test the usability and effectiveness of Netty. We plan to test the effectiveness of the tool in a fourth-year course in formal methods and in a circuit design course.

Netty is powerful enough to include program theory such as the one in [6]. Instead of having to translate the code into another language, it would be desirable to execute it directly. We currently intend to use Scheme to implement the execution of expressions that have been refined to a program. It would also be desirable to advance the capabilities of the type-checker. Improvements would include the accommodation of union-types, and checking some non-trivial expression equality. Currently Netty has a concrete grammar that restricts the available theories. It would be desirable to allow the user complete freedom of theory, including how operators are defined.

Currently, Netty presents suggestions simply in the order that laws are entered in a law file. Ideally, suggestions should be ordered with the most likely steps at the top of the list. In addition, suggestions can be extended to patterns of steps. There are several benefits to this such as allowing faster proving and a more intuitive progression through the proof while maintaining full formality. This could be absolutely invaluable to learning. The implementation of this sort of pattern detection will likely involve a machine learning algorithm. Currently the most suitable kinds of techniques appear to be reinforcement learning techniques. We would need to use data from our empirical studies as training data for the algorithm.

Acknowledgment

We would like to thank Robert Will for laying the groundwork for the tool and initial discussion. We would also like to thank David Kordalewski for discussing ideas and for assistance in implementing Netty. We would also like to thank the referees for helpful suggestions that have improved the paper.

References

[1] R. Back. *Structured derivations: a unified proof style for teaching mathematics*. Formal Aspects of Computing, vol. 22, no. 5, pp. 629-661, Sep 2010.

[2] B. Beckert, R. Hähnle, and Peter H. Schmitt *Verification of Object-Oriented Software. The KeY Approach*. Springer-Verlang, Berlin, 2007.

[3] A.D. Brucker, L. Brügger, M.P. Krieger, and B. Wolff. *HOL-TestGen 1.5.0 User Guide*. ETH Zurich, Technical Report 670, 2010.

[4] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. Version 8.2, July 2009. ¡http://coq.inria.fr/refman/¿ 31.08.2011.

[5] F.B. Fitch. *Symbolic Logic*. The Ronald Press Company, New York, 1952.

[6] E.C.R. Hehner. *a Practical Theory of Programming*. Springer, New York, 1993. ¡http://www.cs.utoronto.ca/∼hehner/aPToP¿ 31.08.2011.

[7] E.C.R. Hehner, R.J. Will, L. Naiman, and D. Kordalewski. *The Netty Project*. University of Toronto, 2011. ¡http://www.cs.utoronto.ca/∼hehner/Netty¿ 31.08.2011.

[8] A.Y.C. Lai. *A Tool for A Formal Refinement Method*. MSc Thesis, University of Toronto, January 2000.

[9] L.C. Paulson. *Isabelle: A generic theorem prover*. Springer-Verlag, Berlin, 1994.

[10] N. Shankar, S. Owre, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS Prover Guide*. Version 2.4, November 2001.

[11] R.J. Will. *Constructing Calculations from Consecutive Choices: a Tool to Make Proofs More Transparent*. MSc Thesis, University of Toronto, January 2010.

Balanced Reduction of an IDE-based Spatio-Temporal Model

Kenneth Scerri*, Michael Dewar†, Parham Aram‡, Dean Freestone§ Visakan Kadirkamanathan‡, David Grayden§
*Department of Systems and Control Engineering, University of Malta, Msida, Malta
Email: kenneth.scerri@um.edu.mt
†Department of Applied Physics and Applied Mathematics, Columbia University, New York, USA
‡Department of Automatic Control and Systems Engineering, University of Sheffield, Sheffield, UK
§Department of Electrical and Electronics Engineering, University of Melbourne, Melbourne, Australia

***Abstract*—Spatio-temporal models have the potential to represent a wide variety of dynamic behaviour such as the growth of bacteria, the dispersion of a pollutant or the changing spatial patterns in house prices. Classical methods for the simulation of such behaviours suffer from large computational demands due to their high dimensionality. Recent advances in spatio-temporal modelling have proposed a method based on a state-space representation of the spatio-temporal integro-difference equation. Although the dimension reduction obtained when using this model is significant, it is frequently not sufficient for online computation or rapid simulation. Thus this model is revisited in this work and a method for further dimension reduction based on a balanced realization of the state-space model is developed. The results will show that the computational cost reduction obtained is significant at the expense of a minor loss in accuracy.**

***Keywords*-spatio-temporal simulation, balanced model reduction, integro-difference equation.**

I. Introduction

Various natural phenomena in a wide spectrum of scientific disciplines exhibit complex interactions over both space and time. These interactions are particularly common in biology, ecology, meteorology, epidemiology, physics, environmental science and economics. Broad ranging methods have been used to describe spatio-temporal behaviour. For example, in physics, reaction-diffusion processes have been successfully modelled via Coupled Map Lattices (CML) with parameters estimated directly from data [1], [2]. Geo-statistical spatio-temporal models have also been estimated from data in both ecological applications such as the monitoring of pollution [3] and in meteorology for, among others, the modelling for rainfall [4] and wind behaviour [5], [6]. In epidemiology, Auto-Regressive Moving-Average (ARMA) models have been used to describe the diffusion of fowl-pest diseases [7] while hierarchical Bayesian models have been used to analyze geographic disease rates [8].

A problem common to most spatio-temporal models is their usually high dimensionality leading to large computational demands for simulation and prediction. Certain methods suffer from further limitations such as the need of CML to have data measured on a regular grid; an impractical condition in applications such as meteorology and epidemiology. Moreover, CMLs also require some knowledge of the natural laws involved to propose an adequate model structure. This knowledge is not always at hand when modelling complex behaviour common for ecological or meteorological applications. Finally, even when measurements are taken on a regular grid, it is often required to infer estimates at other locations among the measurement sites. Unfortunately, most modelling strategies do not provide efficient and rigorous methods to perform such spatial interpolations.

A promising mathematical description of spatio-temporal behaviour that has the potential to overcome or minimize the effect of these limitations is the Integro-Difference Equation (IDE) [9], [10]. In this representation, the spatio-temporal dynamics are governed by a convolution integral in space and a difference equation in time, with the spatio-temporal dynamics dictated by a convolution kernel. In some recently proposed representations, the evolving field modelled by the IDE is decomposed into a set of weighted basis functions also used to decompose the convolution kernel [11], [12]. These decompositions allow the approximate representation of the IDE by a finite dimension state-space model. This framework has the advantage of decoupling the number of states from observation locations with the potential of overcoming the dimensionality issues hampering various other models. Moreover, since in the proposed methods the convolution kernel is completely estimated from data, no prior knowledge of the natural laws involved is required. Finally, since the spatio-temporal behaviour is represented by a basis function decomposition, spatial interpolation is both computationally efficient and mathematically sound.

Other recent additions to this IDE-based spatio-temporal model have proposed a method based on spectral analysis to identifying an adequate number of basis functions to represent some measured behaviour [13], [14]. Nevertheless, the models obtained may still suffer from large computational demands when the spatial bandwidth and/or the spatial domain under investigation are large. Thus in this work a method from systems theory is used to further reduce the dimensionality of the model obtained. The errors introduced by this order reduction procedure will be given analytically and shown experimentally.

The remainder of this paper is organised as follows.

In Section II the state-space representation of the IDE is presented, followed by the dimension reduction procedure and the errors introduced in Section III. Section IV expands on the advantages of the proposed method based on a synthetic example. Finally, Section V gives some concluding remarks and possible future enhancements.

II. STATE-SPACE REPRESENTATION OF THE IDE

Consider the spatially continuous, temporally discrete spatio-temporal process $z(\mathbf{s},t) \in \mathbb{R} : \mathbf{s} \in \mathcal{S} \subset \mathbb{R}^n, t \in \mathbb{Z}^+$ where $n = \{1,2,3,\ldots\}$, $\mathcal{S}$ is a fixed spatial domain and $\mathbf{s}$ and t are spatial and temporal indexes, respectively.

Definition 1. The temporally Markovian, spatially homogeneous, time invariant, Gaussian spatio-temporal IDE is given by

$$z(\mathbf{s},t) = \int_{\mathcal{S}} k(\mathbf{s}-\mathbf{r}) z(\mathbf{r},t-1)\,\mathrm{d}\,\mathbf{r} + \eta(\mathbf{s},t) \tag{1}$$

where $k(\mathbf{s}-\mathbf{r}) : \mathbb{R}^n \to \mathbb{R}$ is a spatially homogeneous convolution kernel and $\eta(\mathbf{s},t)$ is a zero mean stationary Gaussian noise process with covariance Σ_η given by

$$\Sigma_\eta = \mathrm{COV}[z(\mathbf{s},t), z(\mathbf{s}+\mathfrak{s}, t+\mathfrak{t})] = \begin{cases} \lambda(\mathfrak{s}) & \text{if } \mathfrak{t}=0 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Remarks.

1) *The spatio-temporal dynamics of the system are governed by the shape of the convolution kernel. The choice on the space of the kernel is dictated by the process under investigation; for example, simple reaction-diffusion processes can be modelled by Gaussian kernels [9].*
2) *Although a spatially homogenous, time-invariant kernel will be considered here, heterogenous, time-varying kernels can easily be incorporated in the representation as shown in [11].*
3) *The temporal dynamics are here limited to first order Markovian, this assumption can be lifted by including higher order terms with different convolution kernels.*

The stochastic process $z(\mathbf{s},t)$ is observed via a number of identical noisy sensors located at $\{\mathbf{s}_i,\ i=1,2,\ldots,n_y\}$ to obtain the data-set $Y = \{y_t,\ t=1,2,\ldots,T\}$ where $y_t = [y(\mathbf{s}_1,t)\ y(\mathbf{s}_2,t)\ \ldots\ y(\mathbf{s}_{n_y},t)]^\top$. Each sensor can be characterized by the spatial convolution

$$y(\mathbf{s}_i,t) = \int_{\mathcal{S}} h(\mathbf{s}_i-\mathbf{r}) z(\mathbf{r},t) d\,\mathbf{r} + v(t) \tag{3}$$

where $h(\mathbf{s}_i-\mathbf{r})$ is the spatial response of the sensors used and $v(t)$ is a zero mean white Gaussian noise process uncorrelated with $\eta(\mathbf{s},t)$.

The direct computational representation of the stochastic process $z(\mathbf{s},t)$ is intractable due to the continuous nature of the spatial domain. To overcome this problem, [11], [12] have suggested a method based on basis function decompositions of the stochastic process, the convolution kernel, the spatial response of the sensor and the noise covariance to obtain an approximate discrete state-space representation of the IDE. In these methods, the state-space dimension is given by the number of basis functions used to decompose the dynamic field. In [13] a method based on spectral analysis and multi-dimensional extensions of Shannon sampling theorem have been used to obtain an initial estimate of the number, position and parameters of the basis functions used for the decomposition. Joint estimation of the stochastic process and the convolution kernel from noisy data can then be performed by a variety of methods such as the dual Kalman filter [15], the Expectation Maximization (EM) algorithm [16] or in a Bayesian setting, by a 2-stage Gibbs sampler [17].

Such a state-space representation of the IDE requires that the Assumptions 1 and 2 are satisfied.

Assumption 1 (Spatial Low-Pass Response)**.** The spatio-temporal process $z(\mathbf{s},t)$ must exhibit a spectral low-pass behaviours, that is:

$$Z(\boldsymbol{\nu},t) \approx 0\ \forall\ t,\ \ \boldsymbol{\nu} \notin \mathcal{V} \tag{4}$$

where $Z(\boldsymbol{\nu},t)$ is the Fourier transforms of $z(\mathbf{s},t)$ and $\mathcal{V} = [0,\nu_c]^n$, with ν_c being the spatial cut-off frequency.

Assumption 2 (Spatial Semi-Compact Support)**.** The spatio-temporal process $z(\mathbf{s},t)$ must be semi-compactly supported, that is:

$$z(\mathbf{s},t) \approx 0\ \forall\ t,\ \ \mathbf{s} \notin \mathcal{S}\,. \tag{5}$$

Remarks.

1) *Assumption 1 implies that the spatio-temporal process must exhibit some spatial smoothness. Such a condition is generally satisfied by most practical processes.*
2) *Assumption 2 implies that the spatial domain under observation must be finite, again a condition that is usually satisfied in most spatio-temporal studies.*

Using the basis function approximations

$$z(\mathbf{s},t) \approx \sum_{j=1}^{n_x} \langle z(\mathbf{s},t), \phi_{x_j}(\mathbf{s})\rangle \phi_{x_j}(\mathbf{s}) = \mathbf{x}(t)^\top\, \boldsymbol{\phi}_x(\mathbf{s}) \tag{6}$$

$$k(\mathbf{s}) \approx \sum_{j=1}^{n_\theta} \langle k(\mathbf{s}), \phi_{\theta_j}(\mathbf{s})\rangle \phi_{\theta_j}(\mathbf{s}) = \boldsymbol{\theta}^\top\, \boldsymbol{\phi}_\theta(\mathbf{s}) \tag{7}$$

$$h(\mathbf{s}) \approx \sum_{j=1}^{n_\vartheta} \langle h(\mathbf{s}), \phi_{\vartheta_j}(\mathbf{s})\rangle \phi_{\vartheta_j}(\mathbf{s}) = \boldsymbol{\vartheta}^\top\, \boldsymbol{\phi}_\vartheta(\mathbf{s}) \tag{8}$$

$$\lambda(\mathbf{s}) \approx \sum_{j=1}^{n_\varrho} \langle \lambda(\mathbf{s}), \phi_{\varrho_j}(\mathbf{s})\rangle \phi_{\varrho_j}(\mathbf{s}) = \boldsymbol{\varrho}^\top\, \boldsymbol{\phi}_\varrho(\mathbf{s}) \tag{9}$$

where

$$\begin{aligned} \mathbf{x}(t) &= [\langle z(\mathbf{s},t), \phi_{x_1}(\mathbf{s})\rangle \quad \dots \quad \langle x(\mathbf{s},t), \phi_{x_{n_x}}(\mathbf{s})\rangle]^\top \\ \boldsymbol{\theta} &= [\langle k(\mathbf{s}), \phi_{\theta_1}(\mathbf{s})\rangle \quad \dots \quad \langle k(\mathbf{s}), \phi_{\theta_{n_\theta}}(\mathbf{s})\rangle]^\top \\ \boldsymbol{\vartheta} &= [\langle h(\mathbf{s}), \phi_{\vartheta_1}(\mathbf{s})\rangle \quad \dots \quad \langle h(\mathbf{s}), \phi_{\vartheta_{n_\vartheta}}(\mathbf{s})\rangle]^\top \\ \boldsymbol{\varrho} &= [\langle \lambda, (\mathbf{s}) \phi_{\varrho_1}(\mathbf{s})\rangle \quad \dots \quad \langle \lambda(\mathbf{s}), \phi_{\varrho_{n_\varrho}}(\mathbf{s})\rangle]^\top \\ \boldsymbol{\phi}_i(\mathbf{s}) &= [\phi_{i_1}(\mathbf{s}) \ \dots \ \phi_{i_{n_i}}(\mathbf{s})]^\top \end{aligned} \tag{10}$$

and $\phi_i(\mathbf{s})$ are some chosen basis functions, an approximate state-space representation of the IDE with known error bounds is given by Theorem 1.

Theorem 1. *Using the spatially discrete representations (6) to (9) with Assumptions 1 and 2 satisfied, the stochastic IDE of Definition 1 and the observation equation (3) can be approximated by the finite dimension state-space model*

$$\mathbf{x}(t+1) = A(\boldsymbol{\theta})\,\mathbf{x}(t) + \mathbf{w}(t) \tag{11}$$

and

$$\mathbf{y}(t) = C(\boldsymbol{\vartheta})\,\mathbf{x}(t) + \mathbf{v}(t) \tag{12}$$

where

$$A(\boldsymbol{\theta}) = \Psi^{-1} \int_{\mathcal{S}} \boldsymbol{\phi}_x(\mathbf{s})\,\boldsymbol{\theta}^\top\,\Xi_\theta(\mathbf{s}) d\,\mathbf{s} \tag{13}$$

$$\Psi = \int_{\mathcal{S}} \boldsymbol{\phi}_x(\mathbf{s})\,\boldsymbol{\phi}_x(\mathbf{s})^\top d\,\mathbf{s} \tag{14}$$

$$\Xi_\theta(\mathbf{s}) = \int_{\mathcal{S}} \boldsymbol{\phi}_\theta(\mathbf{s}-\mathbf{r})\,\boldsymbol{\phi}_x(\mathbf{r})^\top d\,\mathbf{r} \tag{15}$$

$$C(\boldsymbol{\vartheta}) = \begin{pmatrix} \boldsymbol{\vartheta}^\top\,\Xi_\vartheta(\mathbf{s}_1) \\ \vdots \\ \boldsymbol{\vartheta}^\top\,\Xi_\vartheta(\mathbf{s}_{n_y}) \end{pmatrix}$$

$$\Xi_\vartheta(\mathbf{s}) = \int_{\mathcal{S}} \boldsymbol{\phi}_\vartheta(\mathbf{s}-\mathbf{r})\,\boldsymbol{\phi}_x(\mathbf{r})^\top d\,\mathbf{r} \tag{16}$$

$$\mathbf{w}(t) \sim \mathcal{N}(0, \Sigma_w) \tag{17}$$

with

$$\Sigma_w = \Psi^{-1} \int_{\mathcal{S}} \boldsymbol{\phi}_\varrho(\mathbf{s}) \varrho(\mathbf{s})^\top \Xi_\varrho(\mathbf{s}) d\,\mathbf{s}\,\Psi^{-\top} \tag{18}$$

$$\Xi_\varrho(\mathbf{s}) = \int_{\mathcal{S}} \boldsymbol{\phi}_\varrho(\mathbf{s}-\mathbf{r})\,\boldsymbol{\phi}_x(\mathbf{r})^\top d\,\mathbf{r} \tag{19}$$

and $\mathbf{v}(t) \sim \mathcal{N}(0, \Sigma_v)$ *with* $\Sigma_v = \sigma_v I_{n_y}$*; with errors in the approximation of* $z(\mathbf{s},t)$ *given by*

$$\epsilon_z = |z(\mathbf{s},t) - \mathbf{x}(t)^\top\,\boldsymbol{\phi}_x(\mathbf{s})| \leqslant \epsilon'_z \int_{\mathbb{R}^n : \boldsymbol{\nu} > \boldsymbol{\nu}_c} \Phi_x(\boldsymbol{\nu}) d\,\boldsymbol{\nu} \tag{20}$$

where

$$\epsilon'_z = \sup_{\mathbb{R}^n : \boldsymbol{\nu} > \boldsymbol{\nu}_c} |Z(\boldsymbol{\nu}) \Phi_x^{-1}(\boldsymbol{\nu})| \tag{21}$$

Remarks.

1) *Proof of Theorem 1 is given in [13].*
2) *The given error bounds assume that a closed form solution for (14), (15), (16) and (19) exist. Such a condition is satisfied by Gaussian basis functions.*
3) *The state evolution equation (11) of the state-space model of Theorem 1 can be rewritten as*

$$\mathbf{x}(t+1) = A(\boldsymbol{\theta})\,\mathbf{x}(t) + B\grave{\mathbf{w}}(t) \tag{22}$$

where $\grave{\mathbf{w}}(t)$ *is a zero-mean Gaussian white noise process with covariance* $\Sigma_{\grave{w}} = I_{n_x}$ *and* B *being the Cholesky decomposition of* Σ_w*, that is* $BB^* = \Sigma_w$ *where* B^* *denotes the conjugate transpose of* B*.*

The computation cost of simulating a spatio-temporal process using this model depends on the state-space model dimension. Based on Theorem 1, good approximations can only be obtained if the full spatial extent and the full bandwidth are considered. This often results in computationally expensive models. A choice to limit the system bandwidth can be taken, but this results in the spatial smoothing of the field estimates and predictions. As an alternative, Section III presents a dimension reduction method based on a balanced realization of the state-space model.

III. Dimension Reduction

Balanced reduction methods of state-space models rely on two steps:

1) The original state-space model is first transformed into a balanced realisation with the states most effected by noise being also the most observable states.
2) The states that are less effected by noise in the balanced realisation (and therefore also less observable) are removed to obtain an approximate truncated state-space model.

The initial transformation into a balanced realization requires that all states of the state-space model are both perturbable and observable. This allows for a linear transformation of the state-space model into a balanced realization. This balanced model can then be truncated to retain only the perturbable and observable states. The perturbability and observability requirements are ensured if conditions 1 and 2 are satisfied.

Condition 1 (Perturbability)**.** All the states of the state-space model of Theorem 1 are perturbable iff, the matrix

$$P = [B \ AB \ A^2B \ \dots \ A^{n_x-1}B] \tag{23}$$

is of full rank.

Condition 2 (Observability)**.** All the states of the state-space model of Theorem 1 are observable iff, the matrix

$$O = [C \ CA \ CA^2 \ \dots \ CA^{n_x-1}]^\top \tag{24}$$

is of full rank.

Remarks.

1) *For the state-space model of the IDE of Theorem 1, conditions 1 and 2 are easily satisfied by a well spread arrangement of both the sensors and basis functions used to represent the dynamic field. Such an arrangement requires that no two identical sensors or basis functions are positioned at the same spatial location.*

Given that these conditions are satisfied, a balance realization of the state-space model of the IDE is given by Lemma 1.

Lemma 1 (Balance Realization)**.** *If conditions 1 and 2 are satisfied, then a balance realization of the states space model of Theorem 1 is given by:*

$$\breve{\mathbf{x}}(t+1) = \breve{A}\breve{\mathbf{x}}(t) + \breve{B}\grave{\mathbf{w}}(t) \tag{25}$$

and the observation equation

$$\mathbf{y}(t) = \breve{C}\breve{\mathbf{x}}(t) + \mathbf{v}(t) \tag{26}$$

where $\breve{\mathbf{x}}(t) = T\,\mathbf{x}(t)$, $\breve{A} = TAT^{-1}$, $\breve{B} = TB$, $\breve{C} = CT^{-1}$ *and* $T \in \mathbb{R}^{n_x} \times \mathbb{R}^{n_x}$ *is a linear transformation such that the matrices* P *and* O *of the transformed model satisfy* $P^\top P = O^\top O = \Upsilon$, *where*

$$\Upsilon = \mathrm{diag}(\sigma_1,\ \sigma_2\ \ldots\ \sigma_{n_y}) \tag{27}$$

and where $\{\sigma_i, i = 1, 2, \ldots\}$ *are the Hankel singular values of the state-space model with* $\sigma_1 > \sigma_2 > \sigma_3 \ldots$*.*

Remarks.

1) *Proof of Lemma 1 is given for general state-space models in [19].*
2) *Conditions 1 and 2 ensure that the linear transformation* T *exists.*
3) *Standard matrix computation packages provide accurate methods for obtaining the transformation matrix* T*. These methods are mostly based on the contragradient algorithm [18].*

A reduced order model with known error bounds based on the balanced realization of Lemma 1 is given in Theorem 2

Theorem 2. *Using the balanced state-space model of Lemma 1, an approximate reduced order state-space model is given by:*

$$\tilde{\mathbf{x}}(t+1) = \tilde{A}\tilde{\mathbf{x}}(t) + \tilde{B}\tilde{\mathbf{w}}(t) \tag{28}$$

and

$$\mathbf{y}(t) = \tilde{C}\tilde{\mathbf{x}}(t) + \mathbf{v}(t) \tag{29}$$

where $\breve{\mathbf{x}}(t) = [\tilde{\mathbf{x}}(t)^\top \ldots]^\top$, $\grave{\mathbf{w}}(t) = [\tilde{\mathbf{w}}(t)^\top \ldots]^\top$, *with* $\tilde{\mathbf{x}}(t), \tilde{\mathbf{w}}(t) \in \mathbb{R}^{n_r}$ *and*

$$\breve{A} = \begin{pmatrix} \tilde{A} & \cdots \\ \vdots & \ddots \end{pmatrix} \tag{30}$$

$$\breve{B} = \begin{pmatrix} \tilde{B} & \cdots \\ \vdots & \ddots \end{pmatrix} \tag{31}$$

$$\breve{C} = \begin{pmatrix} \tilde{C} & \ldots \end{pmatrix} \tag{32}$$

where $\tilde{A} \in \mathbb{R}^{n_r} \times \mathbb{R}^{n_r}$, $\tilde{B} \in \mathbb{R}^{n_r} \times \mathbb{R}^{n_r}$ *and* $\tilde{C} \in \mathbb{R}^{n_y} \times \mathbb{R}^{n_r}$, *with a maximum error between the impulse responses of the two systems denoted by* ϵ *and given by*

$$\epsilon = 2(\sigma_{n+1} + \sigma_{n+2} + \ldots) \tag{33}$$

Remarks.

1) *Proof of Theorem 2 is given for general state-space models in [19].*
2) *As* $n_r \to n_x$, *the maximum error bound is reduced at the cost of a higher dimensional model and thus higher computational demands.*

IV. Example

To illustrate the advantages and assess the error introduced by the proposed model reduction procedure, a synthetic data-set was generated by the IDE of Definition 1 and the observation process (3) using numerical integration. All functions and parameters of the IDE and the observation equation are as given in Table I.

Function or parameter		Simulation Value
$\mathcal{S}$	$\in$	$[-6, 6]$
t	$\in$	$[0, 10]$
$k(s)$	=	$0.35\exp(-s^2) + 0.2\exp(-(s-1)^2)$
$\lambda(s)$	=	$\delta(s)$
$z(s,0)$	=	$\frac{1}{\sqrt{4\pi}}\exp(\frac{-(s)^2}{4})$
$h(s)$	=	$\frac{1}{\sigma\sqrt{2\pi}}\exp(-\frac{1}{2}\frac{\mathbf{s}^2}{0.7})$
σ_v^2	=	0.01
n_y	=	25 (equally spaced)

Table I
IDE AND OBSERVATION EQUATION FUNCTIONS AND PARAMETERS.

Given the functions and parameters of Table I, a process realisation generated by the IDE is shown in Figures 1.

A first approximate state-space representation of the IDE as given in Theorem 1 is obtained. The state-space and decomposition parameters chosen are as given in Table II.

Function or Parameter		Simulation Value
n_x	=	13
$\phi_x(s)$	=	$\exp\big(-\frac{s}{0.4}\big)$
basis locations for $z(s,t)$	=	$\{-6,\ -5,\ \ldots,\ 6\}$

Table II
STATE-SPACE MODEL ORDER, BASIS FUNCTIONS AND BASIS LOCATIONS.

Based on this representation, the stochastic field shown in Figure 1 is approximated by its discrete basis function reconstruction $\hat{z}_1(s,t)$ shown in Figure 2.

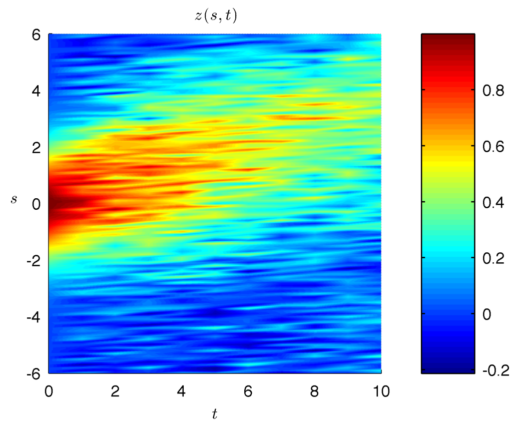


Figure 1. Typical spatio-temporal process $z(\mathbf{s},t)$ generated by first order numerical integration.

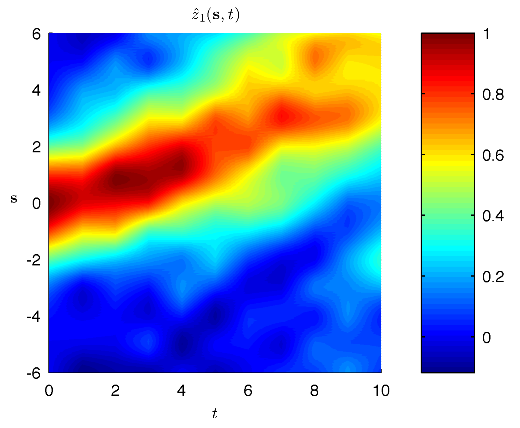


Figure 2. Approximate reconstructed spatio-temporal process $\hat{z}_1(\mathbf{s},t)$ using the discretized model.

Note that the spatially discrete model has a state-space dimension of $n_x = 13$. Thus its system behaviour is captured by a 13×13 matrix which is used in all the computations. Given this state-space representation and the model reduction method of Theorem 2, a reduced order model was obtained with all balanced states with Hankel singular values $\sigma_i < 0.1$ removed. This reduced order model has a state-space dimension of $n_x = 6$ and therefore obtains a significant reduction in computational costs since operations are now performed on 6×6 matrices. This reduced order model generates the approximate field $\hat{z}_2(s,t)$ shown in Figure 3.

The error field $e(s,t)$ showing the error in $\hat{z}_2(s,t)$ introduced by the model reduction procedure when compared to the discretized model field $\hat{z}_1(s,t)$ is shown in Figure 4. This error field indicates that the model reduction procedure has eliminated some higher frequency components but still produced a reasonable approximation to the original stochastic field.

The Root Mean Squared Error (RMSE) between $\hat{z}_1(s,t)$

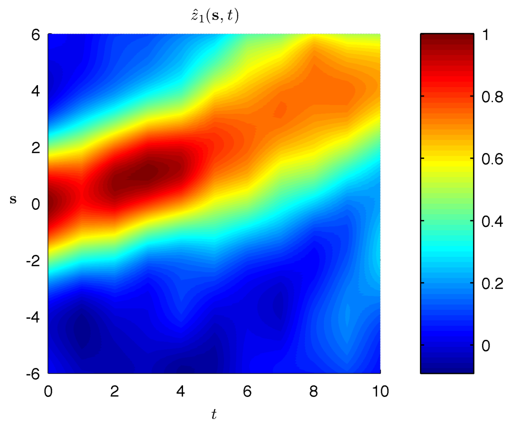


Figure 3. Approximate reconstructed spatio-temporal process $\hat{z}_2(\mathbf{s},t)$ using the reduced order model.

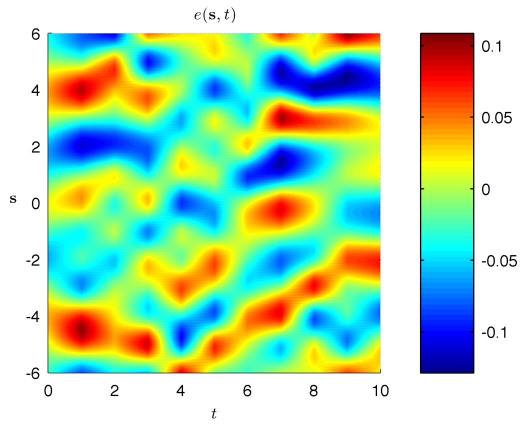


Figure 4. Error field $e(s,t)$ between $\hat{z}_1(s,t)$ and $\hat{z}_2(s,t)$.

and $\hat{z}_2(s,t)$ for the example being considered is 0.051. To verify the repeatability of this result, a Monte Carlo run of 100 different stochastic realizations was performed obtaining the RMSE spread shown in Figure 5, with a mean and standard deviation given by 0.0498 ± 0.0048.

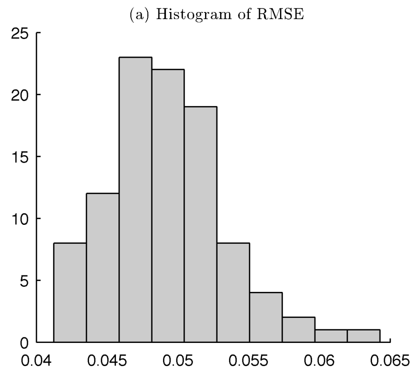


Figure 5. Histogram of RMSE values.

The histogram of Figure 5, indicates the excellent repeatability of the results obtained. Moreover, an error of approximately 0.05 is equivalent to 8% of the average measured observation; a remarkable accuracy considering the 54% reduction in the state-space dimension.

V. Conclusion

Mathematical models of spatio-temporal phenomena are continuously gaining in popularity in various scientific fields. Such models are fundamental for mathematical simulation and therefore prediction of spatio-temporal behaviour. However, these models are frequently severely hampered by the high computation demands of most spatio-temporal simulation methods. Thus in this paper a simulation method based on balanced model reduction of an IDE-based spatio-temporal model is given. The simulation results show the ability of the proposed method to represent spatio-temporal behaviour accurately with significant reductions in the computational costs.

Further work is currently being carried out on this computational method to enhance its applicability to varied applications. While a non-linear growth term, as required in biomedical and ecological applications, has already been included [14], heterogeneous and time-varying implementations have still to be developed. Moreover, applying this computational method to various engineering applications, such as fluid dynamics and mechanical structure analysis, requires the inclusion of boundary conditions. Such additions are also being investigated.

References

[1] S. Billings, L. Guo, and H. Wei, “Identification of coupled map lattice models for spatio-temporal patterns using wavelets,” *International Journal of System Science*, vol. 37, no. 14, pp. 1021–1038, 2006.

[2] D. Coca and S. Billings, “Analysis and reconstruction of stochastic coupled map lattice models,” *Physics Letters A*, vol. 315, pp. 61–75, 2003.

[3] P. Guttorp and P. Sampson, “Methods for estimating heterogeneous spatial convariance functions with environmental applications,” in *Handbook of statistics*, G. Patil and C. Rao, Eds. Elsevier Science, 1994, vol. 12, pp. 661–689.

[4] A. Amani and T. Lebel, “Lagrangian kriging for the estimation of Sahelian rainfall at small time steps,” *Journal of Hydrology*, vol. 192, pp. 125–157, 1997.

[5] X. de Luna and M. Genton, “Predictive spatio-temporal models for spatially spase environmental data,” *Statica Sinica*, vol. 15, pp. 547–568, 2005.

[6] T. Gneiting, “Nonseparable, stationary covariance functions for space-time data,” *Journal of the American Statistical Association*, vol. 97, pp. 590–600, 2002.

[7] R. Martin and J. Oeppen, “The identification of regional forecasting models using space-time correlation functions,” *Transactions of the Institute of British Geographers*, vol. 66, pp. 95–118, 1975.

[8] L. Waller, B. Carlin, H. Xia, and A. Gelfand, “Hierarchical spatio-temporal mapping of disease rates,” *Journal of the American Statistical Association*, vol. 92, pp. 607–617, 1997.

[9] M. Kot, M. Lewis, and P. van den Driessche, “Dispersal data and the spread of invading organisms,” *Ecology*, vol. 77, pp. 2027–2042, 1996.

[10] G. Storvik, A. Frigessi, and D. Hirst, “Stationary space time Gaussian fields and their time autoregressive representation,” *Statistical Modelling*, vol. 2, pp. 139–161, 2002.

[11] C. Wikle, “A kernel-based spectral model for non-Gaussian spatio-temporal processes,” *Statistical Modelling: An International Journal*, 2002.

[12] M. Dewar, K. Scerri, and V. Kadirkamanathan, “Data driven spatiotemporal modelling using the integro-difference eqaution,” *IEEE Transactions on Signal Processing*, vol. 57, no. 1, pp. 83–91, 2009.

[13] K. Scerri, M. Dewar, and V. Kadirkamanathan, “Estimation and model selection of an IDE-based spatio-temporal models,” *IEEE Transactions on Signal Processing*, vol. 57, no. 2, pp. 482–492, 2008.

[14] D. Freestone, P. Aram, M. Dewar, K. Scerri, D. Grayden, and V. Kadirkamanathan, “A data-driven framework for neural field modelling,” *NeuroImage*, vol. 56, pp. 1043–1058, 2011.

[15] E. Wan and R. van der Merwe, “The unscented Kalman filter,” in *Kalman filtering and neural networks*, S. Haykin, Ed. Wiley, 2001, ch. 7.

[16] S. Gibson and B. Ninness, “Robust maximum-likelihood estimation of multivariable dynamic systems,” *Automatica*, vol. 41, pp. 1667–1682, 2005.

[17] C. Robert and G. Casella, *Monte Carlo Statistical Methods*. Springer, 2004.

[18] A. Laub, M. Heath, C. Paige, and R. Ward, “Computation of system balancing transformations and other applications of simultaneous diagonalization algorithms,” *IEEE Transactions on Automatic Control*, vol. 32, no. 2, pp. 115–122, 1987.

[19] K. Zhou and J. Doyle, *Essentials of Robust Control*. Prentice Hall, 1998.

A Tool for Signal Probability Analysis of FPGA-Based Systems

Cinzia Bernardeschi[1], Luca Cassano[1], Andrea Domenici[1] and Paolo Masci[2]
[1] Department of Information Engineering, University of Pisa, Italy
[2] Department of Electronic Engineering and Computer Science, Queen Mary University of London, United Kingdom
Email: {cinzia.bernardeschi, luca.cassano, andrea.domenici}@ing.unipi.it, paolo.masci@eecs.qmul.ac.uk

***Abstract*—We describe a model of Field Programmable Gate Array based systems realised with the Stochastic Activity Networks formalism. The model can be used (i) to debug the circuit design synthesised from the high level description of the system, and (ii) to calculate the signal probabilities and transition densities of the circuit design, which are parameters that can be used for reliability analysis, power consumption estimation and pseudo random testing. We validate the developed model by reproducing the results presented in other studies for some representative combinatorial circuits, and we explore the applicability of the proposed model in the analysis of real-world devices by analysing the actual implementation of a circuit for the generation of Cyclic Redundancy Check codes.**

***Keywords*-FPGA, Signal Probability, Simulation, Transition Density.**

I. Introduction and Related Works

Field Programmable Gate Array (FPGA) devices are widely used components in many different application fields, including safety-critical systems. Especially in embedded and mobile applications, the assessment of such quality factors as power consumption and reliability is of fundamental importance. It has been shown that these factors may be estimated in terms of *signal probability* [1], [2], [3], [4], that can be defined as the fraction of clock cycles in which a given signal is high [5]. Another useful parameter is *transition density*, i.e., the fraction of clock cycles in which a signal makes a transition [6]. Other applications of signal probabilities are *soft error rate* estimation [7] and *random testing* [8]. Soft error rate is the error rate due to Single Event Upsets, i.e., errors caused by radiations, that are a major threat to system reliability. In random testing, test patterns are generated at random to cover as many as possible fault modes of the system. The statistical distribution of the test patterns may be weighted according to the input signal probabilities to optimise the coverage.

The computation of signal probabilities may rely on either an analytical or a simulative approach. With analytical models, exact values of signal probabilities can be computed, but the computation is NP-hard in the general case [9], so it is usually necessary to resort to heuristic approximations. With a simulative approach, a model of the system is fed with inputs whose values reflect the expected statistical properties, and the simulated output signals are recorded and analysed to evaluate the resulting properties.

In this paper, we present a model of FPGA circuit execution that can be used to calculate the signal probabilities and transition densities of a given FPGA design, starting from the signal probabilities of the inputs. The model is based on the formalism of *Stochastic Activity Networks* (SAN) [10] and it has been developed with the Möbius tool [11].

In FPGA systems, a high-level design is implemented with the configurable logic blocks made available by a given FPGA chip. In order to attain a realistic model and satisfactory accuracy of the analysis, the proposed model represents the FPGA system at this implementation level.

The model is implemented by a simulator that takes as input a description of the system to be simulated and a few configuration parameters, including the signal probabilities of the inputs, the number of simulated clock cycles etc. The simulator generates input vectors according to the specified signal probabilities of the inputs and the results are collected and analysed using the features of the underlying Möbius environment.

In the rest of this paper, the FPGA technology (Section II) and the SAN formalism (Section III) are introduced, then the formal model of FPGA circuit execution is presented (Section IV) and a case study is shown as a proof of concept (Section V). Conclusions and future work are in Section VI.

II. The FPGA Technology

An FPGA is an array of programmable logic blocks, interconnected through a programmable routing architecture and communicating with the output through programmable I/O pads [12]. The programming of an FPGA device consists in downloading a programming code, called *bitstream*, in its configuration memory, that determines the hardware structure of the system to be implemented in the FPGA, and thus the functionality performed by the system.

The logic blocks may be simple combinatorial/sequential functions, such as lookup tables, multiplexers and flip-flops, or more complex structures such as memories, adders, and microcontrollers. The routing architecture in an FPGA consists of wires and programmable switches that form the desired connections among logic blocks and I/O pads. Finally, the I/O architecture is composed of I/O pads disposed along the perimeter of the device, each one implementing one or more communication standards.

An FPGA system is described at the *Register-Transfer Level (RTL)* in terms of high-level registers and logic functions, independent of their implementation on a particular device.

An RTL specification is usually given in a hardware definition language (HDL), such as VHDL or Verilog. At the *netlist* level, a system is described in terms of its actual implementation, targeted at a particular device and composed of the logic blocks made available by the device, such as lookup tables and flip-flops.

The implementation of an FPGA-based application involves three main phases: (i) the RTL level design is specified with schematics or with a textual description in a HDL; (ii) after an FPGA chip has been selected, a chip-specific tool synthesises the RTL description into a *netlist*, i.e., a textual description of the network implementing the design; and (iii) the bitstream is generated from the netlist.

III. The SAN Formalism

Stochastic Activity Networks [10] are an extension of the Petri Nets (PN). SANs are directed graphs with four disjoint sets of nodes: *places*, *input gates*, *output gates*, and *activities*. The latter replace and extend the *transitions* of the PN formalism. The topology of a SAN is defined by its input and output gates and by two functions that map input gates to activities and pairs (*activity*, *case*) (see below) to output gates, respectively. Each input (output) gate has a set of *input* (*output*) places.

Each SAN activity may be either *instantaneous* or *timed*. Timed activities represent actions with a duration affecting the performance of the modelled system, e.g., message transmission time. The duration of each timed activity is expressed via a *time distribution* function. Any instantaneous or timed activity may have mutually exclusive outcomes, called *cases*, chosen probabilistically according to the *case distribution* of the activity. Cases can be used to model probabilistic behaviours. An activity *completes* when its (possibly instantaneous) execution terminates.

As in PNs, the state of a SAN is defined by its *marking*, i.e., a function that, at each step of the net's evolution, maps the places to non-negative integers (called the *number of tokens* of the place). SANs enable the user to specify any desired enabling condition and firing rule for each activity. This is accomplished by associating an *enabling predicate* and an *input function* to each input gate, and an *output function* to each output gate. The enabling predicate is a Boolean function of the marking of the gate's input places. The input and output functions compute the next marking of the input and output places, respectively, given their current marking. If these predicates and functions are not specified for some activity, the standard PN rules are assumed.

The evolution of a SAN, starting from a given marking μ, may be described as follows: (i) The instantaneous activities enabled in μ complete in some unspecified order; (ii) if no instantaneous activities are enabled in μ, the enabled (timed) activities become *active*; (iii) the completion times of each active (timed) activity are computed stochastically, according to the respective time distributions; the activity with the earliest completion time is selected for completion; (iv) when an activity (timed or not) completes, one of its cases is selected according to the case distribution, and the next marking μ' is computed by evaluating the input and output functions; (v) if an activity that was active in μ is no longer enabled in μ', it is removed from the set of active activities.

Graphically, places are drawn as circles, input (output) gates as left-pointing (right-pointing) triangles, instantaneous activities as narrow vertical bars, and timed activities as thick vertical bars. Cases are drawn as small circles on the right side of activities. Gates with default (standard PN) enabling predicates and firing rules are not shown.

A. The Möbius Tool

Möbius [11] is a popular software tool that provides a comprehensive framework for model-based evaluation of system dependability and performance. The Möbius tool introduces *shared variables* and *extended places* as extensions to the SAN formalism. Shared variables are global objects that can be used to exchange information among modules. Extended places are places whose marking is a complex data structure instead of a non-negative integer. Enabling predicates and input and output functions of the gates are specified as C++ code.

SAN models can be composed by means of *Join* and *Rep* operators. Join is used to compose two or more SANs. Rep is a special case of Join, and is used to construct a model consisting of a number of replicas of a SAN. Models composed with Join and Rep interact via *place sharing*.

Properties of interest are specified with *reward functions*. A reward function specifies how to measure a property on the basis of the SAN marking. There are two kinds of reward functions: *rate reward* and *impulse reward*. Rate rewards can be evaluated at any time instant. Impulse rewards are associated with specific activities and they can be evaluated only when the associated activity completes. Measurements can be conducted at specific time instants, over periods of time, or when the system reaches a steady state.

IV. Description of the Model

The model is split into a number of modules that interact through place sharing. Each module identifies a different logical component of the FPGA: *System Manager* co-ordinates the logical components; *Input Vector* models the signals applied to the input pins; *Combinatorial Logic* models the memoryless circuits; *Sequential Logic* models the storage elements.

The communication among modules reflects the logical connections of the real FPGA components. The logical connections are specified in a *connectivity matrix*, which is a parameter of the model. This way, the logical connections are not hardwired in the SAN models, and can be set up from a text file generated with software tools, such as the Xilinx ISE tool [13], on the basis of the specification of the FPGA.

The overall FPGA system model is shown in Figure 1. The models of the logical components are represented with labelled dark boxes, and their composition is obtained through the *Join* and *Rep* operators. Each model, except *System Manager*, is obtained by composing a number of replicas of customisable template models. Each replica is uniquely identified by an

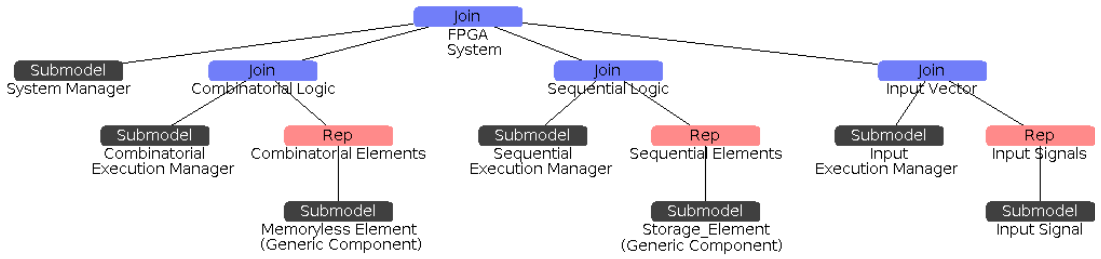


Fig. 1. SAN model of the FPGA.

integer number. The co-ordination between *System Manager* and replicas is accomplished through *Execution Manager*s.

A. System Manager

The System Manager module orchestrates the activity of the other modules of the system according to the following steps: (i) an *input vector*, i.e., an n-tuple of the input signal values, is applied to the input lines; (ii) the combinatorial part of the system is executed; (iii) the clock tick arrives and the sequential part of the system is executed. These steps are repeated until all input vectors have been applied. Steps (ii) and (iii) are repeated until a steady state is reached.

The SAN model of the *System Manager* is shown in Figure 2. The state of the modelled FPGA is given by the marking of three shared places (`input_lines`, `output_lines`, and `internal_lines`), which are vectors that encode the value of the signals on the input, output, and internal lines of the FPGA. Information on the occurrence of transitions on the lines are also maintained in the model with the shared places `input_trans`, `output_trans`, and `internal_trans`. The state includes also two flags: `steady_state_flag`, whose marking reports if the model has reached a steady state; `error_flag`, whose marking reports if the model reaches abnormal execution conditions, e.g., instability of the combinatorial circuit. These flags can be used by analysts and developers to check the consistency of the model specification and to detect potential design problems in the FPGA.

The initial marking of *System Manager* is the following: places `input_lines`, `output_lines`, and `internal_lines` are set according to an initial state of the system; place `signal_length` contains a number of tokens equal to the number of input vectors that will be applied; place `p0` contains one token; all other places hold zero tokens.

Initially, the instantaneous activity `apply_inputs` is enabled because `p0` contains one token. When `apply_inputs` completes, the application of an input vector is triggered by activating module *Input Vector* (Section IV-B). The activation of the *Input Vector* module is obtained by moving the token stored in `p0` into the shared place `sp0_0`. When the application of the input vector completes, module *Input Vector* moves a token into `sp1_1`, and the instantaneous activity `executeCC` of the *System Manager* becomes enabled.

When activity `executeCC` completes, the execution of the combinatorial elements of the model starts by activating module *Combinatorial Logic* (Section IV-C). The activation of this module is obtained by moving the token stored in `sp1_1` into the shared place `sp2_0`. When the execution of the combinatorial elements completes, the *Combinatorial Logic* module moves a token into `sp3_1`, thus enabling the timed activity `executeSC` of the *System Manager*.

When the timed activity `executeSC` completes, a clock tick has arrived, and the execution of the sequential elements starts by activating module *Sequential Logic* (the token stored in `sp3_1` is moved into `sp4_0`). When the execution of the sequential elements completes, the *Sequential Logic* module (Section IV-C) moves a token into `sp5_1`. If the system has reached a steady state, a token is also moved into `steady_state_flag`.

At this point, the instantaneous activity `finalise` is enabled. When `finalise` completes, the marking of the model is updated according to the following three cases: (i) `signal_length` holds more than one token; in this case, the marking of `p0` is incremented by one; this marking triggers the application of a new input vector; (ii) `signal_length` holds zero tokens and the system state is not steady (i.e., `steady_state_flag` contains zero tokens); in this case, a token is moved into `sp1_1`; this marking triggers a new execution of the combinatorial and sequential parts of the model; (iii) `signal_length` has zero tokens and `steady_state_flag` has one token; in this case, the system has reached the final steady state and the execution terminates; a token is moved into `p3` and no activity will be further enabled in the model.

B. Input Vector

The *Input Vector* module applies an input vector to the input lines. The elements of the input vector are generated according to the signal probability of the corresponding signal. The total number of input lines is a model parameter (N_{in}).

The module consists of a manager sub-module, which co-ordinates the concurrent execution of activities in the model, and a number of customised template model replicas, each of which applies an input value to an input line.

1) Execution Manager: This sub-module co-ordinates the parallel execution of the *Input Signal* replicas. The SAN model is shown in Figure 3(a).

In the model, all places initially contain zero tokens, except the shared places that model the FPGA state (`input_lines`, `output_lines`, and `internal_lines`); places `sp0` and `sp1` coincide (through renaming in the Join operator) with

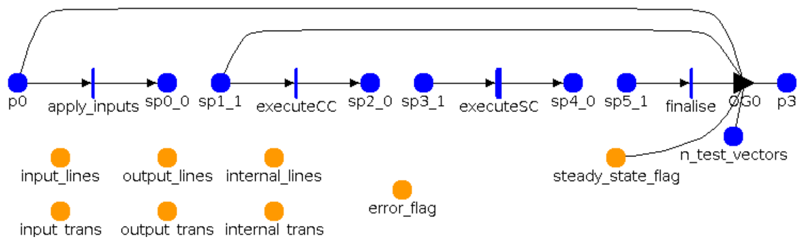
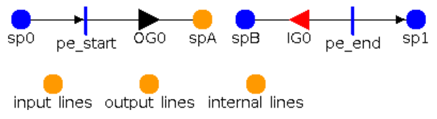
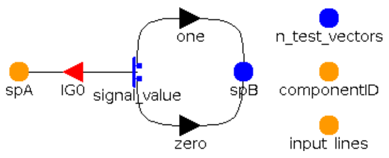


Fig. 2. SAN Model of the System Manager.

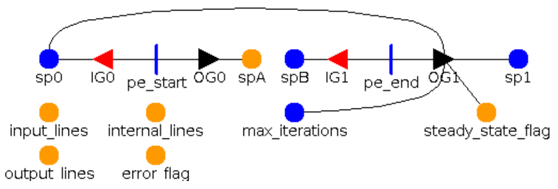


(a) Execution Manager.

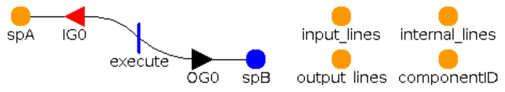


(b) Input Signal.

Fig. 3. Sub-modules of the Input Vector SAN model.



(a) Iterative Execution Manager.



(b) Generic Component.

Fig. 4. Sub-modules of the Combinatorial and Sequential Logic SAN model.

places `sp0_0` and `sp1_1` of *System Manager*, and are thus shared between the two sub-modules. Places `spA` and `spB` are shared with the *Input Signal* sub-modules; `spA` is a vector of N_{in} Booleans; one token in position i encodes a $true$ value for the corresponding element, and triggers the i-th replica of *Input Signal*.

Activity `pe_start` is enabled when *System Manager* moves a token into `sp0`. When the activity completes, the parallel execution of the *Input Signal* replicas starts: gate `OG0` is executed, and N_{in} tokens are moved in the shared vector `spA` (one token for each element of the vector). When the parallel execution of the *Input Signal* replicas concludes, the shared place `spB` contains N_{in} tokens, and the input gate `IG0` enables `pe_end`, which completes immediately. When `pe_end` completes, all tokens stored in `spB` are removed, and one token is moved into `sp1` to signal *System Manager* that the input lines have been updated with the input vector.

2) Input Signal: This sub-module models the application of a signal value at an input line. The SAN model is shown in Figure 3(b). We exploit the semantics of the case probabilities to specify the signal value in terms of the probability of having a logical zero or a logical one.

The *Input Signal* sub-module is replicated N_{in} times, in order to have one sub-module instance for each input line. Replicas have unique identifiers, which are used to associate each replica to an input line.

All places of the model are initially empty, except `componentID`, whose marking specifies the identifier of the sub-module instance. Activity `signal_value` of the sub-module with identifier i is enabled when the *Execution Manager* moves a token in the i-th position of `spA`. When `signal_value` of replica i completes, one of the two output gates is executed to apply a signal value to input line i, and a token is added to element i of `spB`.

C. Combinatorial and Sequential Logic

The *Combinatorial Logic* and the *Sequential Logic* modules define the execution of the memoryless elements and the storage elements, respectively, of the FPGA.

Similarly to *Input Vector*, both models consist of a manager sub-module and a number of customised template model replicas, each of which models the functionalities of an elementary component in the FPGA (either a memoryless element or a storage element).

1) Iterative Execution Manager: This sub-module coordinates the parallel execution of the sub-module replicas. This module is an iterative version of the *Execution Manager* sub-module used in *Input Vector*. This iterative version is used to repeatedly activate the parallel execution of the components until either the modelled elements reach a steady state, or a maximum number of iterations has been performed.

The SAN model of the *Iterative Execution Manager* is shown in Figure 4(a). In the model, all places initially contain zero tokens, except the shared places that model the FPGA state (`input_lines`, `output_lines`, and `internal_lines`), and `max_iterations`, whose marking specifies the maximum number of iterations needed to complete the execution of the modelled elements. In the case of combinatorial elements, the maximum number of iterations depends on the interconnection among elements; in the case of sequential elements, the number of iterations is always one.

We describe the *Iterative Execution Manager* in conjunction with the *Combinatorial Elements* module, as the same description applies also to the *Sequential Elements* module.

In the model of the *Iterative Execution Manager*, shared places `sp0` and `sp1` coincide (through renaming in the Join operator) with the shared places `sp2_0` and `sp3_1` of *System Manager*, and are thus shared between the two submodules. Places `spA` and `spB` are shared with the *Combinatorial* submodules. Moving one token in position i of `spA` encodes a $true$ value for the element in position i, and triggers the execution of the i-th replica of *Input Signal*. The total number of combinatorial elements is a model parameter (N_c).

Activity `pe_start` is enabled when *System Manager* moves a token into `sp0` and `max_iterations` contains at least one token. When the activity completes, the parallel execution of the combinatorial elements starts: gate `OG0` is executed, the marking of `max_iterations` is decremented by one, and N_c tokens are moved in the shared vector `spA` (one token for each element of the vector). When the parallel execution of the combinatorial elements concludes, the shared place `spB` contains N_c tokens, and the input gate `IG0` enables `pe_end`, which completes immediately. When `pe_end` completes, all tokens stored in `spB` are removed, and one token is moved into `sp1` to signal *System Manager* that the execution of the combinatorial elements has completed.

2) Combinatorial and Sequential Elements: These elements are modelled through a customisable SAN model, denominated `Generic_Component` (Figure 4(b)).

The set of combinatorial (sequential) elements is obtained by replicating N_c (N_s) times the *Generic Component* model, where N_c (N_s) is the total number of combinational (sequential) elements. Each replica has a unique identifier, used to specify the functionality of each model instance.

All places of the model are initially empty, except `componentID`, which specifies the replica identifier, and `input_lines`, `output_lines`, and `internal_lines`, representing the current system state. The instantaneous activity `execute` of replica i is enabled when the *Iterative Execution Manager* moves a token in the i-th position of `spA`. When the `execution` activity of replica i completes, the function specified in gate `OG0` is executed, and a token is added to `spB`.

V. Analysis

This section presents the results obtained through the simulation of the proposed SAN models using the Möbius [11] tool. The goal of the presented analysis is two-fold: (i) to validate the developed model for the FPGA, by reproducing the results presented in other studies for some representative combinatorial circuits; (ii) to explore the applicability of the proposed model in the analysis of real-world devices, by analysing the actual implementation of a circuit for the generation of Cyclic Redundancy Check (CRC) codes. CRC is a widely used error-detection scheme used in data communication systems to contrast communication failures due to the unreliable nature of the physical links between devices. When data needs to be reliably transmitted over an unreliable link, the sending device includes a CRC field in the transmitted message; this way, the receiving device can check if the received message is damaged and, in such case, arrange for a message retransmission.

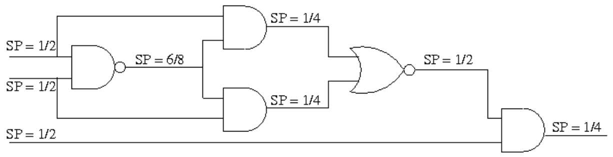


Fig. 5. Example of combinatorial circuit used for validation.

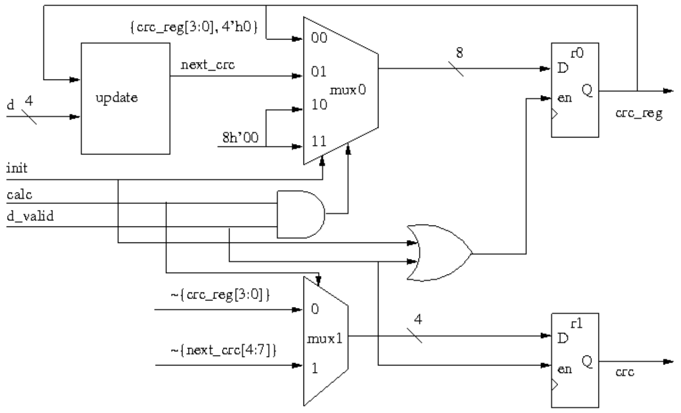


Fig. 6. A circuit to generate CRC code (adapted from [15]).

A. Validation of the Model

To validate our model, we considered the combinatorial circuits presented in various related works and we checked that we were able to reproduce the same results. Let us consider, as a representative case, the combinatorial circuit of Figure 5, which has a reconvergent fanout. The analytical results for signal probability are reported in [14] (we show such results on the upper side of the lines).

The signal probabilities computed with our model correspond with those reported in [14]; specifically, after 10000 simulation runs, we obtained an average relative error of $2 \cdot 10^{-4}$, never exceeding $6 \cdot 10^{-4}$. This cross-validation exercise reinforced our confidence in the correct definition of the model.

B. Analysis of a Circuit for CRC generation

As a case study, we consider the FPGA implementation of a circuit for the generation of IEEE 802.3 CRC codes [15]. A simplified schematic of a 4-bit data bus circuit that generates 8-bit CRC codes is shown in Figure 6. In the figure, `d` is the 4-bit data bus, `init`, `calc`, and `d_valid` are control signals, and `update` is a combinatorial network that computes the next state for the output register `r0`.

The Verilog code for the circuit, which is publicly available from the Xilinx web site, was compiled into a netlist with the Xilinx ISE tool [13]. The resulting netlist has 8 input signals, 12 output signals, 20 matching I/O buffers, 17 LUTs, and 19 flip-flops. We modelled the netlist according to the method described in Section IV and we used the model to compute the signal probabilities and transition densities of the signals on the internal lines of the circuit. The experiments have been set up to reproduce the signal values during a CRC calculation: the control pins `load_init` and `reset` are always low; `calc` is always high; `d_valid` switches between high and low levels at each clock cycle; input pins `d[3:0]` are high with probability P_i, where i is the index of the input pin.

The measurements have been obtained as follows: (i) A simulation run consists in applying a number n_r of consecutive test vectors, where r identifies the run; the test vectors elements are generated with a uniform probability distribution, assuming independence between elements and between vectors; (ii) for each signal i, we defined two reward functions: p_{ri}, which returns, for each run r, the number of clock cycles in which the i-th signal is high; t_{ri}, which returns, in each run r, the number of clock cycles in which the signal makes a transition. The signal probability P_{ri} and transition density T_{ri} of signal i for run r are then computed by dividing p_{ri} and t_{ri}, respectively, by n_r.

We obtain a quantitative assessment of signal probability and transition density for every line of the circuit. The number of test vectors used for the experiment is $n_r = 48$, which corresponds to the length of an IEEE 802.3 address field. The number of simulation runs needed to obtain a confidence level of 95% for this circuit was between $5K$ and $10K$. The time needed to execute $1K$ simulation runs of the model on a 2.67 GHz Intel Core i5 was about 2 minutes.

Some results are shown in Figure 7. The plots shown in the figure report (on the y axis) the value of signal probability and transition density of three internal lines connected to multiplexer `mux0` when varying (on the x axis) the signal probability of the input lines `d[3:0]`. In order to simplify the presentation of the results, in the experiments we imposed that the signal probability varies identically on all input lines.

VI. Conclusions and Future Work

A general model for the execution of FPGA circuits at the netlist level has been defined with the SAN formalism and a simulator has been developed with the Möbius tool. The proposed model is suitable (i) to debug the actual FPGA circuit design synthesised from the high level description of the system and (ii) to compute signal probabilities and transition densities of the design. It is worth noting that, even if the analysis has focused on FPGA systems, the model is applicable to general sequential circuits.

As further work, we intend to extend the model to study system reliability with fault-injection techniques, and we intend to exploit the capabilities of the SAN formalism to model both independent or correlated faults.

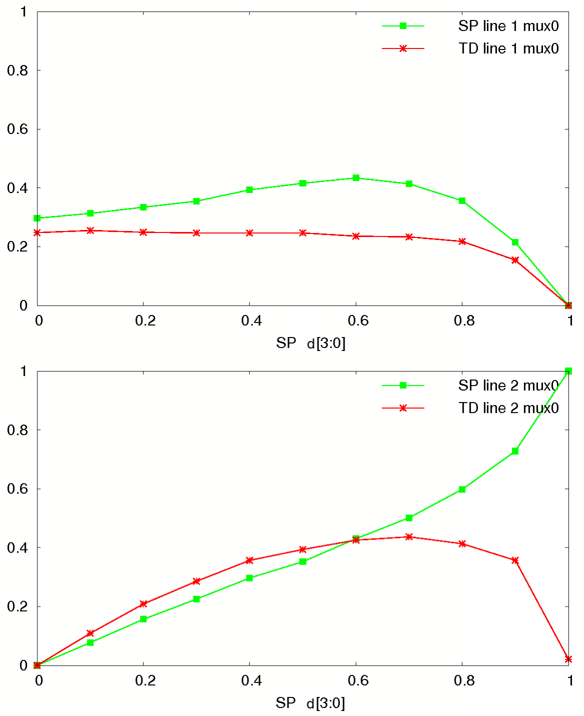


Fig. 7. Example of results for signal probability and transition density of two input lines of mux0 for input data signal probability 0.5.

References

[1] F. N. Najm, "A survey of power estimation techniques in vlsi circuits," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, no. 4, pp. 446–455, december 1994.

[2] D. Franco *et al.*, "Signal probability for reliability evaluation of logic circuits," *Microelectronics Reliability*, vol. 48, no. 8-9, pp. 1586–1591, 2008.

[3] J. T. Flaquer *et al.*, "Fast reliability analysis of combinatorial logic circuits using conditional probabilities," *Microelectronics Reliability*, vol. 50, no. 9-11, pp. 1215–1218, 2010.

[4] H. Chen and J. Han, "Stochastic computational models for accurate reliability evaluation of logic circuits," in *Proceedings of the 20th symposium on Great lakes symposium on VLSI*, ser. GLSVLSI '10. ACM, 2010, pp. 61–66.

[5] K. Parker and E. McCluskey, "Analysis of logic circuits with faults using input signal probabilities," *IEEE Transactions on Computers*, vol. C-24, no. 5, pp. 573–578, may 1975.

[6] V. Saxena *et al.*, "Estimation of state line statistics in sequential circuits," *ACM Transactions on Design Automation of Electronic Systems*, vol. 7, no. 3, pp. 455–473, july 2002.

[7] H. Asadi *et al.*, "Soft error susceptibility analysis of SRAM-based FPGAs in high-performance information systems," *IEEE Transactions on Nuclear Science*, vol. 54, no. 6, pp. 2714–2726, dec 2007.

[8] J. Savir, "Distributed generation of weighted random patterns," *Computers, IEEE Transactions on*, vol. 48, no. 12, pp. 1364–1368, Dec. 1999.

[9] B. Krishnamurthy and I. Tollis, "Improved techniques for estimating signal probabilities," *IEEE Transactions on Computers*, vol. 38, no. 7, pp. 1041–1045, Jul. 1989.

[10] W. H. Sanders and J. F. Meyer, "Stochastic activity networks: formal definitions and concepts." New York, NY, USA: Springer-Verlag New York, Inc., 2002, pp. 315–343.

[11] G. Clark *et al.*, "The Möbius modeling tool," in *9th Int. Workshop on Petri Nets and Performance Models*. Aachen, Germany: IEEE Computer Society Press, September 2001, pp. 241–250.

[12] I. Kuon *et al.*, "Fpga architecture: Survey and challenges," *Foundations and Trends in Electronic Design Automation*, vol. 2, 2008.

[13] "ISE Design Suite Software Manuals and Help," http://www.xilinx.com/support/documentation/sw_manuals, 2010.

[14] M. Al-Kharji and S. Al-Arian, "A new heuristic algorithm for estimating signal and detection probabilities," in *Proceedings. Seventh Great Lakes Symposium on VLSI, 1997*, mar 1997, pp. 26–31.

[15] C. Borrelli, "IEEE 802.3 Cyclic Redundancy Check," application note: Virtex Series and Virtex-II Family, XAPP209 (v1.0), March 23, 2001, Xilinx, Inc.

Formal Verification of Parameterized Multi-agent Systems Using Predicate Diagrams*

Cecilia E. Nugraheni
Informatics Department
Parahyangan Catholic University
Bandung, Indonesia
Email: cheni@unpar.ac.id

***Abstract*—This paper presents a formal diagram-based verification technique for multi-agent systems. A multi-agent system is a collection of intelligent agents that interact with each others and work together to achieve a goal. We view multi-agent systems as parameterized systems which are systems that consist of several similar processes whose number is determined by an input parameter. The motivation of this work is that by treating multi-agent systems as parameterized systems, the specification and verification processes can be done in the same way regardless of the number of agents involved in the multi-agent systems. In this paper, we show how predicate diagrams* can be used to represent the abstractions of parameterized multi-agent systems described by specifications written in TLA*. The verification process is done by integrating deduction verification and algorithmic techniques. The correspondence between the original specification and the diagram is established by non-temporal proof obligations; whereas model checker SPIN is used to verify properties over finite-state abstractions.**

Keywords-multi-agent systems; parameterized systems; verification; predicate diagrams*; TLA*; TLA+.

I. Introduction

A multi-agent system is understood as a collection of intelligent agents, in this case are software or programs that interact with each others and work together to achieve a goal. Sycara [22] said that ”Agent-based systems technology has generated lots of excitement in recent years because of its promise as a new paradigm for conceptualizing, designing, and implementing software systems. This promise is particularly attractive for creating software that operates in environments that are distributed and open, such as the internet.” Because of this promise, many researches on multi-agent systems have been conducted. Most of these researches concentrate on the specification and verification of the agents' behaviors and the coordination among agents.

In this work, we focus on multi-agent systems which consist of several similar processes. Having this property, a multi-agent system can be viewed as a parameterized system, which is a system that consists of several similar processes whose number is determined by an input parameter. Many interesting systems are of this form. One of them is mutual exclusion algorithms for an arbitrary number of processes wanting to use a common resource. The motivation of this work is that by treating multi-agent systems as parameterized systems, the specification and verification processes can be done more easily. Both processes are expected to be done in the same way regardless of the number of agents involved in the multi-agent systems.

Verification consists of establishing whether a system satisfies some properties, that is, whether all possible behaviors of the system are included in the properties specified. It is common to classify the approaches to formal verification into two groups, which are the deductive approach and the algorithmic approach. The deductive approach is based on theorem proving and typically reduces the proof of a temporal property to a set of proofs of first-order verification conditions. The most popular algorithmic verification method is model checking [6], [7], [21]. Although this method is fully automatic for finite-state systems, it suffers from the so-called state-explosion problem.

The need for a more intuitive approach to verification leads to the use of diagram-based formalisms. Basically, a diagram is a graph whose nodes represent sets of system states and whose edges represent the transition of the systems. Diagram-based approach combines the advantages of deductive and algorithmic approach, which are the process is goal-directed, incremental and can handle infinite-state systems.

In the context of parameterized systems, to provide methods for the uniform verification of such systems is a challenging problem. One solution of this problem is to treat or to represent a family of objects as a single syntactic object. This technique is called parameterization.

In [20], a diagram-based verification for parameterized systems is proposed. The diagrams, which are called predicate diagrams*, are variants of diagrams proposed by Cansell et al. In [5], they presented a class of diagrams called predicate diagrams and showed how to use the diagrams in formal verification. In [20], a little modification of the definition of the original predicate diagrams is made, in particular the definition related to the actions. Instead of actions, the new approach concentrates only on parameterized actions which are actions of the form $A(k)$. This form of actions is usually used in modeling actions of a particular process in the system. TLA* [17] is used to formalize this approach and TLA+ [13] style is used to write specifications.

This paper is structured as follows. Section II explains briefly the formal specification of parameterized systems in TLA*. Section III describes the definition and the use of predicate diagrams* in the verification of parameterized systems. The next section describes the aplication of this approach on a case study which is block world problem. Discussion about the result and some related works are given in Section V. Finally, conclusion and future work will be given in Section VI.

II. Pameterized Systems Specification

This work is restricted to a class of parameterized systems that are interleaving and consist of a finitely, but arbitrarily, discrete components. Let M denote a finite and non-empty set of processes running in the system. A parameterized system can be described as a formula of the form:

$$parSpec \equiv \forall k \in M : Init(k) \wedge \Box[Next(k)]_{v[k]} \wedge L(k).$$

where

- $Init$ is a state predicate describing the global initial condition,
- $Next(k)$ is an action characterizing the next-state relation of a process k,
- v is a state function representing the variables of the system and
- $L(k)$ is a formula stating the liveness conditions expected from the process or subsystem k.

III. Predicate Diagrams*

Basically, a predicate diagram* is a finite graph whose nodes are labeled with sets of (possibly negated) predicates, and whose edges are labeled with actions as well as optional annotations. This section gives a brief description of predicate diagrams*. For a detail explanation for predicate diagrams*, the readers may consult [20].

A. Definition

It is assumed that the underlying assertion language contains a finite set $\mathcal{O}$ of binary relation symbols $\prec$ that are interpreted by well-founded orderings. For $\prec \in \mathcal{O}$, its reflexive closure is denoted by $\preceq$. We write $\mathcal{O}^=$ to denote the set of relation symbols $\prec$ and $\preceq$ for $\prec \in \mathcal{O}$.

Definition 1. Assume given two finite sets $\mathcal{P}$ and $\mathcal{A}$ of state predicates and parameterized action names. A predicate diagram* $G = (N, I, \delta, o, \zeta)$ over $\mathcal{P}$ and $\mathcal{A}$ consists of :

- a finite set $N \subseteq 2^{\overline{\mathcal{P}}}$ of nodes where $\overline{\mathcal{P}}$ denotes the set of literals formed by the predicates in $\mathcal{P}$,
- a finite set $I \subseteq N$ of initial nodes,
- a family of δ_A where $A \in \mathcal{A}$ of relations $\delta_{A \subseteq N \times N}$; we also denote by δ the union of the relations δ_A, for $A \in \mathcal{A}$ and write $\delta^=$ to denote the reflexive closure of the union of these relations,
- an edge labeling o that associates a finite set $\{(t_1, 1), \ldots, (t_k, k)\}$, of terms t_i paired with a relation $\prec_i \in \mathcal{O}^=$ with every edge $(n, m) \in \delta$, and
- a mapping $\zeta : \mathcal{A} \to \{\text{NF,WF,SF}\}$ that associates a fairness condition with every parameterized action in $\mathcal{A}$; the possible values represent no fairness, weak fairness, and strong fairness.

A parameterized action $A \in \mathcal{A}$ can be taken at node $n \in N$ iff $(n, m) \in \delta_A$ holds for some $m \in N$. The set of nodes where A can be taken is denoted by $En(A)$.

A *run* of a predicate diagram* is a sequence of triples, $\rho = (s_0, n_0, A_0)(s_1, n_1, A_1) \ldots$ where s_i is a state, $n_i \in N$ is a node and $A_i \in \mathcal{A} \cup \{\tau\}$ (τ denotes stuttering transition). A *trace* through a predicate diagram*, $\sigma = s_0 s_1 \ldots$, is defined as the set of those behaviors that correspond to fair runs satisfying the node and edge labels.

B. Verification using predicate diagrams*

The verification process using predicate diagrams* is done in two steps. The first step is to find a predicate diagram* that conforms to the system specification. Theorem 1 is used to prove the conformance.

Theorem 1. Let $G = (N, I, \delta, o, \zeta)$ be a predicate diagram* over $\mathcal{P}$ and and $\mathcal{A}$, let $parSpec = Init \wedge \Box[\exists k \in M : Next(k)]_v \wedge \forall k \in M : L(k)$ be a parameterized system. If all the following conditions hold then G conforms to $parSpec$:

1) For all $n \in I$, $\models Init \to n$.
2) $\models\!\approx n \wedge [\exists k \in M : Next(k)]_v \to$
$$n' \vee \bigwedge_{(m, A(k)) : (n,m) \in \delta_{A(k)}} \langle \exists k \in M : A(k) \rangle_v \wedge m'$$
3) For all $n, m \in N$ and all $(t, \prec) \in o(n, m)$:
   a) $\models\!\approx n \wedge m' \wedge \bigwedge_{A(k) : (n,m) \in \delta_{A(k)}} \langle \exists k \in M : A(k) \rangle_v \to t' \prec t$.
   b) $\models\!\approx n \wedge [\exists k \in M : Next(k)]_v \wedge n' \to t' \preceq t$.
4) For every parameterized action $A(k) \in \mathcal{A}$ such that $\zeta(A(k)) \neq$ NF:
   a) If $\zeta(A(k)) =$ WF then
   $\models parSpec \to \text{WF}_v(\exists k \in M : A(k))$.
   b) If $\zeta(A(k)) =$ SF then
   $\models parSpec \to \text{SF}_v(\exists k \in M : A(k))$.
   c) $|\!\approx n \to$ ENABLED $\langle \exists k \in M : A(k) \rangle_v$ holds whenever $n \in En(A(k))$.
   d) $|\!\approx n \wedge \langle \exists k \in M : A(k) \rangle_v \to m'$ holds for all $n, m \in N$ such that $(n, m) \notin \delta_{A(k)}$.

The second step of the verification process is to prove that all traces through a predicate diagram* satisfy some property F. In this step, a diagram predicate* is viewed as a finite transition system. As a finite transition system, its

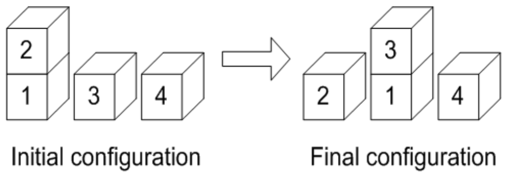


Figure 1. An example of block world problem.

runs can be encoded in the input language of standard model checkers such as SPIN [3].

IV. The block world : a case study

A. Problem statement

One of the most famous planning domains in artificial intelligence is the block world problem. This problem can be briefly described as follows [23]: given a set of cubes (blocks) on a table and two types of robots whose task is to change the vertical stacks of blocks, from the initial configuration into a new different configuration. It is assumed that only one block may be moved at a time: it may either be placed on the table or placed atop another block. Any block that is, at a given time, under another block cannot be moved. Each robot has different capability: one robot is only capable of ’freeing’ one block from another, while the other is only capable of ’moving’ one block and putting it on top of another.

As illustration, Figure 1 shows an example of this problem. There are four blocks on the table. Each block is labeled with a number. The left-hand side represents the initial state of the world or initial configuration, whereas the right-hand side represents the goal (the final configuration) to be achieved.

B. Specification

From the problem statement, it is clear that the agents or the robots in the block world problem are not homogeneous. In order to model this problem as a parameterized system as required, one simple solution is taken, which is: (1) every robot is associated with an integer stating its type and (2) the capabilities of every robot type’s are stated in a separate action. This is done in order to give an impression that the robots have different capabilities. Only now, an additional condition or precondition is added to each action. The homogeneity requirement is still guaranteed by using the same $Next$ action for each agent.

The specification for the block world problem is given in Figure 2. In this specification we use a set of positive integers, $Blocks$, to represent the collection of the blocks and an array ag_type to identify the agent’s type.

To represent the state or configuration of the blocks, we use an array whose elements are pairs of non-negative integers. Each element represents the condition of a block.

$$
\begin{aligned}
isDone &\equiv CState = FState\\
move(k) &\equiv \wedge\ \neg isDone \wedge ag\_type[k] = 1\\
&\quad \wedge\ \exists x, y \in Blocks :\\
&\qquad \wedge\ x \neq y\\
&\qquad \wedge CState[x] = \langle 0, 0\rangle \wedge CState[y][2] = 0\\
&\qquad \wedge CState' = [CState \text{ EXCEPT } ![x][1] = y,\\
&\qquad\qquad ![y][2] = x]\\
free(k) &\equiv \wedge\ \neg isDone \wedge ag\_type[k] = 2\\
&\quad \wedge\ \exists x, y \in Blocks :\\
&\qquad \wedge\ x \neq y\\
&\qquad \wedge CState[x][2] = 0 \wedge CState[y][2] = x\\
&\qquad \wedge CState' = [CState \text{ EXCEPT } ![x] = \langle 0, 0\rangle,\\
&\qquad\qquad ![y][2] = 0]\\
Init &\equiv \wedge\ CState = IState \wedge CState \neq FState\\
&\quad \wedge\ \forall k \in M : ag\_type[k] \in \{1, 2\}\\
Next(k) &\equiv Move(k) \vee Free(k)\\
L(k) &\equiv \mathrm{WF}_v(Move(k)) \wedge \mathrm{WF}_v(Free(k))\\
v &\equiv \langle CState\rangle\\
BWorld &\equiv \wedge Init \wedge \Box[\exists k \in M : Next(k)]_v\\
&\quad \wedge \forall k \in M : L(k)
\end{aligned}
$$

Figure 2. Specification for Block World Problem.

For example, if the second element of the array is $\langle 3, 1\rangle$ then it means that the block number 3 is under the block number 2 and the block number 1 is on the block number 2. A special number, 0, is used to represent table or nothing. Thus, the initial configuration of Figure 1 is represented by $\langle\langle 0, 2\rangle, \langle\langle 1, 0\rangle, \langle 0, 0\rangle, \langle 0, 0\rangle\rangle$ and the final configuration is represented by $\langle\langle 0, 3\rangle, \langle 0, 0\rangle, \langle 1, 0\rangle, \langle 0, 0\rangle\rangle$.

Three state arrays are used, which are $IState$, $CState$ and $FState$. Each array is used to represent initial, final, and current configuration, respectively. Current configuration records the last configuration of the system. The goal is achieved whenever $isDone$ is true, which means that the current and the final configuration have the same values.

Action $move(k)$ can be taken only by an agent whose capability is to move a block from the table and put it onto another block. Action $free(k)$ can be taken only by an agent whose capability is to free one block on the top of a stack and put it on the table. Formula $CState' = [CState$ EXCEPT$![x][1] = y, ![y][2] = x]$ in $move(k)$ action means that except the values of $CState[x]$ and $CState[y]$, the values of $CState'$ won’t change after the action is taken.

The values of $Blocks, ag\_type, IState$, and $FState$ depend on the problem instance at hand. If the number of boxes and agent types change, we simply change $Blocks$ and ag_types accordingly. This holds also for the capabilities of the agents. We need only to add precondition to every action as explained.

C. Verification

We take a problem instance in Figure 1 for the verification purpose. The following formulas are added to our specification in Figure 2:

- $Blocks \equiv \{1, 2, 3, 4\}$
- $IState \equiv \langle\langle 0, 2\rangle, \langle 1, 0\rangle, \langle 0, 0\rangle, \langle 0, 0\rangle\rangle$

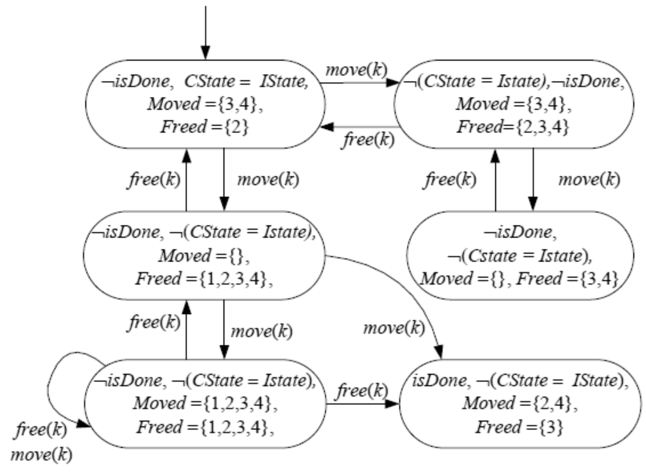

¬isDone, CState = IState, Moved ={3,4}, Freed ={2}
move(k)
free(k)
¬(CState = Istate),¬isDone, Moved ={3,4}, Freed={2,3,4}
free(k)
move(k)
free(k)
move(k)
¬isDone, ¬(CState = Istate), Moved ={}, Freed ={1,2,3,4},
¬isDone, ¬(Cstate = Istate), Moved ={}, Freed ={3,4}
free(k)
move(k)
move(k)
¬isDone, ¬(CState = Istate), Moved ={1,2,3,4}, Freed ={1,2,3,4},
free(k)
isDone, ¬(CState = IState), Moved ={2,4}, Freed ={3}
free(k)
move(k)

Figure 3. Predicate diagrams* for Block World Problem.

- $FState \equiv \langle\langle 0,3\rangle,\langle 0,0\rangle,\langle 1,0\rangle,\langle 0,0\rangle\rangle$

The verification can be stated as to prove the following theorem:

$$BWorld \rightarrow \Box(Init \rightarrow \Diamond isDone).$$

For the first step, we have to find a suitable predicate diagram* for the $BWorld$. Figure 3 depicts one of the possible predicate diagrams* for *BWorld*.

We set $\mathcal{P}$ to contain six predicates. The union of those predicates define the properties of system's states that hold on every node. There are two predicates that are not written explicitly, which are :$k \in \{1,2\}$ and $\forall k \in M : ag\_type \in \{1,2\}$. These predicates hold on every node. We also use two sets $Moved$ and $Freed$ which are sets of integers to indicate the blocks that can be moved or can be freed, respectively. It is assume that the following conditions hold:

- $\forall i \in Blocks : i \in Moved \leftrightarrow (\exists j \in Blocks : i \neq j \wedge CState[i] = \langle 0,0\rangle \wedge CState[j][2] = 0)$
- $\forall i \in Blocks : i \in Freed \leftrightarrow (\exists j \in Blocks : i \neq j \wedge CState[i]\langle j,0\rangle \wedge CState[j][2] = i)$

Using Theorem 1 it can be shown that the predicate diagram* in Figure 3 conforms to the specification in Figure 2. This is done by proving the following formulas:

- $Init \rightarrow \neg isDone \wedge \neg(CState = IState) \wedge Moved = \{3,4\} \wedge Freed = \{2\}$
- $\neg isDone \wedge \neg(CState = IState) \wedge Moved = \{3,4\} \wedge Freed = \{2\} \wedge [\exists \in M : Next(k)]_v \rightarrow$
  $(\neg isDone' \wedge \neg(CState' = IState') \wedge Moved' = \{3,4\} \wedge Freed' = \{2\}) \vee$
  $((\langle \exists k \in M : free(k)\rangle_v \wedge \neg isDone' \wedge \neg(CState' = IState') \wedge Moved' = \{3,4\} \wedge Freed' = \{2,3,4\}) \vee$
  $(\langle \exists k \in M : free(k)\rangle_v \wedge \neg isDone' \wedge \neg(CState' = IState') \wedge Moved' = \{\} \wedge Freed' = \{1,2,3,4\}))$
- $\neg isDone \wedge \neg(CState = IState) \wedge Moved = \{\} \wedge Freed = \{1,2,3,4\} \wedge [\exists \in M : Next(k)]_v \rightarrow$
  $(\neg isDone' \wedge \neg(CState' = IState') \wedge Moved' = \{\} \wedge Freed' = \{1,2,3,4\}) \vee$
  $((\langle \exists k \in M : move(k)\rangle_v \wedge isDone' \wedge \neg(CState' = IState') \wedge Moved' = \{2,4\} \wedge Freed' = \{3\}) \vee$
  $(\langle \exists k \in M : move(k)\rangle_v \wedge \neg isDone' \wedge \neg(CState' = IState') \wedge Moved' = \{1,2,3,4\} \wedge Freed' = \{1,2,3,4\}))$
- $\neg isDone \wedge \neg(CState = IState) \wedge Moved = \{1,2,3,4\} \wedge Freed = \{1,2,3,4\} \wedge [\exists \in M : Next(k)]_v \rightarrow$
  $(\neg isDone' \wedge \neg(CState' = IState') \wedge Moved' = \{1,2,3,4\} \wedge Freed' = \{1,2,3,4\}) \vee$
  $((\langle \exists k \in M : free(k)\rangle_v \wedge (isDone' \wedge \neg(CState' = IState') \wedge Moved' = \{\} \wedge Freed' = \{1,2,3,4\} \vee \neg isDone \wedge \neg(CState = IState) \wedge Moved = \{1,2,3,4\} \wedge Freed = \{1,2,3,4\})) \vee$
  $(\langle \exists k \in M : move(k)\rangle_v \wedge \neg isDone' \wedge (\neg(CState' = IState') \wedge Moved' = \{2,4\} \wedge Freed' = \{3\} \vee \neg isDone \wedge \neg(CState = IState) \wedge Moved = \{1,2,3,4\} \wedge Freed = \{1,2,3,4\}))$
- $(isDone \wedge \neg(CState = IState) \wedge Moved = \{2,4\} \wedge Freed = \{3\}) \wedge [\exists \in M : Next(k)]_v \rightarrow$
  $(isDone' \wedge \neg(CState' = IState') \wedge Moved' = \{2,4\} \wedge Freed' = \{3\})$
- $(\neg isDone \wedge \neg(CState = IState) \wedge Moved = \{3,4\} \wedge Freed = \{2,3,4\}) \wedge [\exists \in M : Next(k)]_v \rightarrow$
  $(\neg isDone' \wedge \neg(CState' = IState') \wedge Moved' = \{3,4\} \wedge Freed' = \{2,3,4\}) \vee$
  $((\langle \exists k \in M : free(k)\rangle_v \wedge (\neg isDone' \wedge CState' = IState' \wedge Moved' = \{3,4\} \wedge Freed' = \{2\})) \vee$
  $(\langle \exists k \in M : move(k)\rangle_v \wedge \neg isDone' \wedge (\neg(CState' = IState') \wedge Moved' = \{\} \wedge Freed' = \{3,4\}))$
- $(\neg isDone \wedge \neg(CState = IState) \wedge Moved = \{\} \wedge Freed = \{3,4\}) \wedge [\exists \in M : Next(k)]_v \rightarrow$
  $(\neg isDone' \wedge \neg(CState' = IState') \wedge Moved' = \{\} \wedge Freed' = \{3,4\}) \vee$
  $((\langle \exists k \in M : free(k)\rangle_v \wedge (\neg isDone' \wedge \neg(CState' = IState') \wedge Moved' = \{3,4\} \wedge Freed' = \{2,3,4\})))$

The next step we encode the diagram in the input language of SPIN. We use 12 variables, which are:

- $action$ and $node$ to indicate the last action taken and the current node,
- $done$ to indicate whether $isDone$ is true or not,
- $cistate$ to indicate whether $CState = IState$ is true or not,
- $m1, m2, m3$, and $m4$ to represent predicate $1 \in Moved$, $2 \in Moved$, $3 \in Moved$ and $4 \in Moved$, respectively, and
- $f1, f2, f3$, and $f4$ to represent predicate $1 \in Freed$, $2 \in Freed$, $3 \in Freed$ and $4 \in Freed$, respectively,

The theorem to be proven is now can be written as $\Box((\neg done \wedge cistate \wedge \neg m1 \wedge \neg m2 \wedge m3 \wedge m4 \wedge \neg f1 \wedge f2 \wedge \neg f3 \wedge \neg f4) \rightarrow \Diamond(done \wedge \neg cistate \wedge \neg m1 \wedge m2 \wedge \neg m3 \wedge m4 \wedge \neg f1 \wedge \neg f2 \wedge f3 \wedge \neg f4))$. Last, by using SPIN we model-checked the resulted transition system. As result, we concluded that the specification satisfies the property we want to prove.

V. Related work

Many works are devoted to the formal specification of multi-agent systems. Most of these works concentrate on the specification of the agents' behaviors and the coordination among agents. Abouaissa et al. [1] presented a formal approach for specification of multi-agent systems. This approach is based on roles and organization notions and high-level Petri nets and is applied on a multi-modal platform associating combined rail-road transportation system. Merayom et al. [16] proposed a formalism called Utility State Machines for specifying e-commerce multi agent systems. Brazier et al. [3] used the DESIRE framework to specify a real-world multi-agent application on a conceptual level. Originally DESIRE is designed for formal specification of complex reasoning systems. Fischer and Wooldridge [9] described first step towards the formal specification and verification of multi-agent systems, through the use of temporal belief logics. This work is closed to the one of Taibi [23]. The similarity is that we use TLA-based to formalize our approaches. However, Taibi's work did not treat the multi -agent systems as parameterized systems.

Besides formal specification, formal verification is also a popular topic in the field of multi-agent systems. Several approaches can be found in [2], [10], [11], [15], [23]. In [11], Giese et al. presented an approach for making complex multi-agent system specifications. Every specification includes a detailed environment model that amenable to verification. The verification process is done by means of simulation and formal verification. Taibi used TLC, the TLA model checker, to verify the specification [23]. Gaud et al. proposed a formal framework based on multi-formalisms language for writing system specification and used abstraction to reduce the state space of the system [10]. Ayed et al. proposed a diagram-based verification by using AUML protocol diagrams for representing multi-agent systems. These diagrams are then translated into event-B language for verification purpose [2]. In [15], Massaci et al. concerned about the use of access control for limiting the agent capability of distributed systems. They presented a prefixed tableau method for the calculus of access control. The calculus was the basis for the development and the verification of an implemented system.

Because diagrams can reflect the intuitive understanding of the systems and their specifications, they are proposed to be used for verification. A diagram can also be seen as an abstraction of the system, where properties of the diagram are guaranteed to hold for the systems as well. In particular, the use of diagrams in verification of distributed systems can be found; for example in [5] the author proposed the use of predicate diagrams, introduced in [4], for analyzing a self-stabilizing algorithm.

VI. Conclusion

We have shown how a multi-agent system can be viewed and thus can be formally specified and verified as a parameterized system. In particular, we define a general form for specification of parameterized multi-agent systems in TLA*. By considering a case study, we have shown that a multi-agent system whose agents are not homogenous still can be specified as a parameterized system.

In this paper, we have successfully write specification and verify the block world problem. This problem is an example of multi-agent system whose agents are not homogeneous. In order to fulfill the homogeneity requirement, we add preconditions to actions in the specification to guarantee that only the appropriate agent may take a particular action. For verification process we use predicate diagram* to represent the abstractions of the systems. The correspondence between the original specification and the diagram is established by non-temporal proof obligations; whereas model checker SPIN is used to verify properties over finite-state abstractions.

In the context of parameterized systems, there are two classes of properties that may be considered, namely the properties related to the whole processes and the ones related to a single process in the system. It is planned to investigate those properties of the block world problem. The verification will be conducted by using a diagram-based verification called parameterized predicate diagrams [18], [19].

References

[1] H. Abouaissa, J.C. Nicolas, A. Benasser, and E. Czesnalowicz. *Formal specification of multi-agent systems: approach based on meta-models and high-level Petri nets - case study of a transportation system*. Proceedings of IEEE International Conference on Systems, Man and Cybernetics, Vol.5, pp. 429-434, 2002.

[2] B. Ayed and F. Siala. *Event-B based Verification of Interaction Properties In Multi-Agent Systems*. Journal of Software, Vol. 4, No. 4, pp. 357-364, June 2009

[3] F. Brazier, B. Dunin-Keplicz, N.R. Jennings, and J. Treur. *Formal Specification of Multi-Agent Systems: a Real World Case*. Proceedings of the First International Conference on Multi-Agent Systems, ICMAS'95, MIT Press, Cambridge, MA, 1995, pp.25-32.

[4] D. Cansell, D. Méry, and S. Merz. *Predicate diagrams for the verification of reactive systems*. In 2nd Intl. Conf. on Integrated Formal Methods IFM 2000, vol. 1945 of Lectures Notes in Computer Science, pp. 380-397, 2000. Springer-Verlag.

[5] D. Cansell, D. Méry, and S. Merz. *Formal analysis of a self-stabilizing algorithm using predicate diagrams*. GI Jahrestagung (1) 2001: 39-45.

[6] E.M. Clarke and E.A. Emerson. *Characterizing correctness properties of parallel programs using fixpoints*. *International Colloquim on Automata, Languages and Programming*. Vol. 85 of Lecture Nodes in Computer Science, pp. 169-181, Springer-Verlag, July, 1980.

[7] E.M. Clarke and E.A. Emerson. *Design and synthesis of synchronization skeletons using branching time temporal logic*. Workshop on Logic of Programs, Yorktown Heights, NY. Vol. 131 of Lecture Nodes in Computer Science, pp. 52-71, Springer-Verlag, 1981.

[8] E. A. Emerson and K. S. Namjoshi. *Verification of a parameterized bus arbitration protocol*. Volume 1427 of Lecture Notes in Computer Science, pp. 452–463. Springer,1998.

[9] M. Fisher and M. Wooldridge. *On the Formal Specification and Verification of Multi-Agent Systems*. Int. J. Cooperative Inf. Syst., 1997: 37-66.

[10] N. Gaud. *A Verification by Abstraction Framework for organizational Multi-Agent Systems*. Jung, Michel, Ricci and Petta (eds.): AT2AI-6 Working Notes, From Agent Theory to Agent Implementation, 6th Int. Workshop, May 13, 2008, AAMAS 2008, pp. 67-73, Estoril, Portugal, EU.

[11] H. Giese and F. Klein. *Systematic verification of multi-agent systems based on rigorous executable specifications*. Journal International Journal of Agent-Oriented Software Engineering, Volume 1 Issue 1, pp. 28-62, April 2007.

[12] G. Holzmann. *The SPIN model checker. IEEE Trans*. On software engineering, 16(5):1512-1542. May 1997.

[13] L. Lamport. *The Temporal Logic of Actions. ACM Transactions on Programming Languages and Systems*, 16(3) : 872-923, May 1994.

[14] Z. Manna and A. Pnueli. *Verification of parameterized programs*. In Specification and Validation Methods (E. Borger, ed.), Oxford University Press, pp. 167-230, 1994.

[15] F. Massacci. *Tableau Methods for Formal Verification of Multi-Agent Distributed Systems*. Journal of Logic and Computation, 8(3), 1998.

[16] M.G. Merayo. *Formal specification of multi-agent systems by using EUSMs*. Proceedings of the 2007 international conference on Fundamentals of software engineering, pp. 318-833.

[17] S. Merz. *Logic-based analysis of reactive systems: hiding, composition and abstraction*. Habilitationsschrift. Institut fr Informatik. Ludwig-Maximillians-Universitt, Munich Germany. December 2001.

[18] C.E. Nugraheni. *Predicate diagrams as basis for the verification of reactive systems*. PhD Thesis. Institut fr Informatik. Ludwig-Maximillians-Universitt, Munich Germany. February 2004.

[19] C.E. Nugraheni. *Universal properties verification of parameterized parallel systems*. In Proceeding of the International Confe-rence on Computational Scince and its Applications (ICCSA 2005), Volume 3482 of Lecture Notes in Computer Science, pp. 453-462. Springer, 2005.

[20] C.E. Nugraheni. *Formal Verification of Ring-based Leader Election Protocol using Predicate Diagrams**. IJCSNS Vol. 9. no. 8, pp. 1-8, August 2009.

[21] J.P. Quielle and J. Sifakis. *Specification and verification of concurrent systems in CESAR*. In M. Dezani-Cianzaglini and Ugo Montanari, editors, International Symposium on Programming. Volume 137 of Lecture Notes in Computer Science, pp. 337-350. Springer-Verlag, 1981.

[22] K.P. Sycara. *Multiagent Systems*. AI Magazine, Vol 19, No 2, pp. 79-92, Summer 1998.

[23] T. Taibi. *Formal specification and validation of multi-agent behaviour using TLA+ and TLC model checker*. Int. J. Artificial Intelligence and Soft Computing, Vol. 1, No. 1, pp. 99-113, 2008.

$JClassic^{+}_{\delta\epsilon}$: A Description Logic Reasoning Tool: Application to Dynamic Access Control

Narhimene Boustia
Saad Dahlab University of Blida
Algeria
nboustia@gmail.com

Aicha Mokhtari
USTHB, Algeria
aissani_mokhtari@yahoo.fr

***Abstract*—This paper presents $JClassic^{+}_{\delta\epsilon}$, a description logic with default and exception that is expressive enough to be of practical use, which can reason on default knowledge and handle a ”weakened kind of disjunction”, allowing a tractable subsumption computation. $JClassic^{+}_{\delta\epsilon}$ is an extension of $JClassic_{\delta\epsilon}$, with the connective *lcs*, which has the same properties as the LCS external operation to compute the least common subsumer of two concepts. $JClassic^{+}_{\delta\epsilon}$ is defined with an intensional semantics. We developp this reasoner to define an access control model, where default and exception connectives are used in representation of context to allow authorization. Consideration of context in access control allows definition of dynamic permissions, for example, permissions given to a doctor in a normal context are not the same that we are in an emergency context.**

***Index Terms*—Description Logic; Defaults and Exceptions; Reasoner; disjunction; access control.**

I. Introduction

The purpose of access control models is to assign permissions to users. The most interesting would be to have the ability to set dynamic permissions dynamic, i.e., context-dependent.

Context may be unique, as it may be a relationship between a number of situations such as emergency or epidemic risk. For this, we need connectors that allow us to represent this information. The reasoner $JClassic^{+}_{\delta\epsilon}$ has been developed for this purpose. Unlike the work of Ventos et al. [1], [2], we did not stay at the theoretical level, but rather we implemented the reasoner.

Donini [3] shows that concept disjunction makes subsumption computation co-NP-Complete. However, disjunction is very useful for knowledge representation.

In this paper, we present a decription logic-based system, named $JClassic^{+}_{\delta\epsilon}$, whose set of connectives is the union of $JClassic_{\delta\epsilon}$ connectives and the ”lcs” connective. The ”lcs” connective is a kind of ”weakened disjunction” allowing us to preserve a tractable subsumption computation (subsumption in $JClassic_{\delta\epsilon}$ has been proved correct, complete and tractable in [4]).

The ”lcs” connective has the same properties as the LCS external operation introduced by Borgida et al. [5] which computes the least common subsumer of two concepts. It was introduced by Ventos et al. in Classic to allow disjunction with a reasonable computation [1], [2].

Because of $JClassic_{\delta\epsilon}$ has been given an intensional semantics, $JClassic^{+}_{\delta\epsilon}$ is provided with an intentional semantics (called $\mathcal{CL}^{+}_{\delta\epsilon}$) based on an algebraic approach. For this, we have first to build an equational system which highlights the main properties of the connectives. The equational system allows to define axiomatically the notion of LCS.

In this paper, we first present our system $JClassic^{+}_{\delta\epsilon}$, we give then definition of ”lcs”. We finally illustrate the use of this tool for access control.

In access control, permission are given to user depending on the actual context. The context can be that the default one in our case represented by the default connector (δ), it can be an exception to the current context represented by the connector Exception (ϵ), as it can be a conjunction or a disjunction of several contexts.

To this end, the reasoner $JClassic_{\delta\epsilon}$ has been enriched by the operator of minimum disjontion in order to have a good level of expressiveness with a polynomial complexity.

II. $JClassic^{+}_{\delta\epsilon}$

$JClassic^{+}_{\delta\epsilon}$ is an non monotonic reasoner based on description logic with default and exception [6] which allows us to deal with default and exceptional knowledge.

The set of connectives of $JClassic^{+}_{\delta\epsilon}$ is the union of the set of connectives of $\mathcal{AL}_{\delta\epsilon}$ [6] presented in [4], [7], [8] and the connective ”lcs”.

The connective δ intuitively represents the common notion of default. For instance, having δFly as a conjunct with Animal in the definition of the concept **Bird** states that birds generally fly.

The connective ϵ is used to represent a property that is not present in the description of the concept or of the instance but that should be. For instance, the definition of **Penguin** in $JClassic^{+}_{\delta\epsilon}$ is $Penguin \equiv Bird \sqcap Fly^{\epsilon}$. The Fly^{ϵ} property expresses the fact that fly should be in the definition of Penguin since it is a bird. The presence of Fly^{ϵ} in the definition of Penguin makes it possible to classify Penguin under the concept Bird.

Formally, the subsumption relation uses an algebraic semantics. The main interest of this approach is the introduction of the definitional point of view of default knowledge: from the definitional point of view, default knowledge can be part of concept definition whereas from the inheritance one it is only considered as a weak implication. A map between the definition of concept and its inherited properties is described. This combinating of definitional and inheritance levels improves the classification process. Figure 1 describes the general architecture of our system.

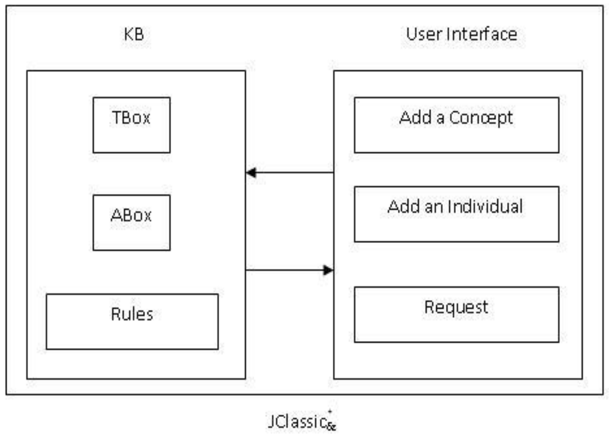


Fig. 1. Architecture of $JClassic_{\delta\epsilon}$

In this section, we first present the syntax of our system, we then give details about its algebraic semantic and we conclude this section by presenting the mechanism of inference in our tools.

A. Syntax of $JClassic^{+}_{\delta\epsilon}$

The set of connectives of $JClassic^{+}_{\delta\epsilon}$ is the union of the set of connectives of $\mathcal{CL}_{\delta\epsilon}$ [6] and the connectives δ and ϵ. $JClassic^{+}_{\delta\epsilon}$ is defined using a set **R** of primitive roles, a set **P** of primitive concepts, the constant $\bot$ (Bottom) and $\top$ (Top) and the following syntax rule (C and D are concepts, P is a primitive concept, R is a primitive role).

$C, D \rightarrow \top$ the most general concept
$\mid \bot$ the most specific concept
$\mid P$ primitive concept
$\mid C \sqcap D$ concept conjunction
$\mid \neg P$ negation of primitive concept (This restriction to primitive concept in the negation is a choice to avoid the untractability)
$\mid \forall r : C$ C is a value restriction on all roles R
$\mid \delta C$ default concept
$\mid C^{\epsilon}$ exception to the concept
$\mid ClcsD$ concept disjunction

δ and ϵ are unary connectives, $\sqcap$ is a binary conjunction connective and $\forall$ enables universal quantification on role values.

B. Semantic of $JClassic^{+}_{\delta\epsilon}$

We endow $JClassic^{+}_{\delta\epsilon}$ with an intentional algebraic semantic denoted $\mathcal{CL}^{+}_{\delta\epsilon}$.

This framework covers the different aspects of the formal definition of concepts and subsumption in our language. The calculating of denotations of concepts in $\mathcal{CL}^{+}_{\delta\epsilon}$ is used in computing subsumption in the algorithm $Sub_{\delta\epsilon}$. $\mathcal{CL}^{+}_{\delta\epsilon}$ allows first to show that $Sub^{+}_{\delta\epsilon}$ is correct and complete and secondly to give a formal characterization of calculation of subsumption used in the implementation of $JClassic^{+}_{\delta\epsilon}$.

1) EQ: an equational system for $JClassic^{+}_{\delta\epsilon}$: In order to serve as the basis for the definition of an algebraic semantics, an equational system EQ is defined. From a descriptive point of view, the calculation of subsumption consists on the comparison of terms through the equational system **EQ**. This system fixes the main properties of the connectives and is used to define an equivalence relation between terms and then to formalize the subsumption relationship.

$\forall A, B, C \in JClassic^{+}_{\delta\epsilon}$:
01: $(A \sqcap B) \sqcap C = A \sqcap (B \sqcap C)$
02: $A \sqcap B = B \sqcap A$
03: $A \sqcap A = A$
04: $\top \sqcap A = A$
05: $\bot \sqcap A = \bot$
06: $(\forall R : A) \sqcap (\forall R : B) = \forall R : (A \sqcap B)$
07: $\forall R : \top = \top$
08: (A lcs B) lcs C = A lcs (B lcs C)
09: A lcs B = B lcs A
10: A lcs A = A
11: A lcs $\top$ = $\top$
12: A lcs $\bot$ = A
13: $(\delta A)^{\epsilon} = A^{\epsilon}$
14: $\delta(A \sqcap B) = (\delta A) \sqcap (\delta B)$
15: $A \sqcap \delta A = A$
16: $A^{\epsilon} \sqcap \delta A = A^{\epsilon}$
17: $\delta\delta A = \delta A$

Axioms 01 to 07 are classical; they concern description logic connectives properties [9], [10]. Axioms 08 to 12 concern the connective ”lcs”. The following ones correspond to $\mathcal{AL}_{\delta\epsilon}$ connectives properties[6], i.e., properties of δ and ϵ connectives.

Descriptive Subsumption:

We denote $\sqsubseteq_d$ for descriptive subsumption. $\sqsubseteq_d$ is a partial order relation on terms. Equality (modulo the axioms of EQ) between two terms is denoted $=_{EQ}$. $=_{EQ}$ is a congruence relation which partitions the set of terms, i.e., $=_{EQ}$ allows to form equivalence classes between terms. We define the descriptive subsumption using the congruence relation and conjunction of concepts as follow:

Definition 1: (Descriptive Subsumption)

Let C and D two terms of $JClassic^{+}_{\delta\epsilon}$, $C \sqsubseteq_d D$, i.e., D subsume descriptively C, iff $C \sqcap D =_{EQ} C$.

From an algorithmic point of view, terms are not easily manipulated through subsumption. We adopt a structural point of view closer to the algorithmic aspect of computing subsumption. This allows us to first formalize calculation of subsumption in the implementation of $JClassic^{+}_{\delta\epsilon}$ and secondly to endow $JClassic^{+}_{\delta\epsilon}$ with an intensional semantics.

To define the subsumption relation between two concepts using their description, we need to compare them. For this, concepts are characterized by a normal form of their properties rather than by the set of their instances.

2) Normal Form of concept: We present in this section the structural point of view for the subsumption in $JClassic^{+}_{\delta\epsilon}$. This point of view has two main advantages: it is very close to the algorithmic aspects and is a formal framework to validate the algorithmic approach which is not the case description graph.

We define a structural concept algebra $\mathcal{CL}^{+}_{\delta\epsilon}$ which is used to give an intensional semantic in which concepts are denoted by the normal form of their set of properties. The structural point of view of subsumption consist then to compare the normal forms derived by applying a homomorphism from set of terms of $JClassic^{+}_{\delta\epsilon}$ to elements of $\mathcal{CL}^{+}_{\delta\epsilon}$.

$\mathcal{CL}^{+}_{\delta\epsilon}$: an intensional semantic for $JClassic^{+}_{\delta\epsilon}$

From the class of CL-algebra, we present a structural algebra $\mathcal{CL}^{+}_{\delta\epsilon}$ which allows to endow $JClassic^{+}_{\delta\epsilon}$ with an intentional semantic.

Element of $\mathcal{CL}^{+}_{\delta\epsilon}$ are the canonical intentional representation of terms of $JClassic^{+}_{\delta\epsilon}$ (i.e., Normal form of the set of their properties). We call an element of $\mathcal{CL}^{=}_{\delta\epsilon}$ normal forms.

Definition of $\mathcal{CL}^{+}_{\delta\epsilon}$ means definition of a homomorphism h which allows to associate an element of $\mathcal{CL}^{+}_{\delta\epsilon}$ to a term of $JClassic^{+}_{\delta\epsilon}$.

Using the equational system, we calculate for each concept a structural denotation which is a single normal form of this concept. The calculation of a normal form from a description of a concept can be seen as a result of term "rewriting" based on the equational system EQ.

The normal form of a concept defined with description T (noted nf(T)) is a couple $\langle t_\theta, t_\delta \rangle$ where t_θ contains strict properties of T and t_δ the default properties of T.

t_θ and t_δ are 3-tuple of the form (π,r,ϵ) with:

π: is a set of primitive concepts in description T.

r: has the form $\langle R, c \rangle$ where :

R: is the name of Role.

c: is the normal form of C, if the description contains the property $\forall R : C$.

ϵ: set of 3-tuple with the form (π, r, ϵ).

Example:

The normal form of concept $A \equiv B \sqcap C \sqcap \delta D$ is:

nf (A) = $(\langle \{B, C\}, \emptyset, \emptyset \rangle, \langle \{B, C, D\}, \emptyset, \emptyset \rangle)$.

Structural Subsumption:

Two terms C and D of $JClassic^{+}_{\delta\epsilon}$ are structurally equivalent iff their normal forms are equal. We denote $\sqsubseteq_s$ for structural subsumption. $\sqsubseteq_s$ is a partial order relation.

The structural equality of two terms of $JClassic^{+}_{\delta\epsilon}$ is noted $=_{CL}$. $=_{CL}$ is a congruence relation as $_{EQ}$ in descriptive subsumption.

We define the structural subsumption using the congruence relation and conjunction of concepts as follow:

Definition 2: **(Structural Subsumption)**

Let C and D two terms of $JClassic^{+}_{\delta\epsilon}$, $C \sqsubseteq_s D$; i.e., D subsume structurally C, iff $C \sqcap D =_{CL} C$.

Theorem 1: **(Equivalency between descriptive subsumption and structural subsumption)**

Let C and D two terms of $JClassic^{+}_{\delta\epsilon}$, $C \sqsubseteq_s D \Leftrightarrow C \sqsubseteq_d D$.

To infer new knowledge in this system, the susbsumption relation is used. In the next section, we outline the subsumption algorithm handling defaults and axceptions named Sub$_{\delta\epsilon}$.

III. Inference in $JClassic^{+}_{\delta\epsilon}$

There are several reasoning services to be provided by a DL- system. We concentrate our work on the following basic ones, which are Classification of concepts (TBox) and instance checking (ABox). These two services basically use the subsumption relation.

A. The Subsumption Relation

Borgida [5] defines the subsumption based on a set theoretic interpretation as follow: "The concept C subsume D, if and only if the set of instances of C include or is equal to a set of instances of D".

However, the general principle of computing subsumption between two concepts is to compare their sets of properties, not their sets of instances.

For this, we use an intensional semantics which is closer to the algorithmic aspects of computing subsumption, and this by defining a normal form of description called descriptive normal form.

Algorithm of Computing Subsumption $Sub_{\delta\epsilon}$

$Sub_{\delta\epsilon}$ is an algorithm of computing subsumption of the form Normalization- Comparison. It is **consists** of two steps, first, the normalization of description, and then a syntactic comparison of the obtained normal forms.

Let C and D be two terms of $JClassic^{+}_{\delta\epsilon}$. To answer the question "Is C subsumed by D?" we apply the following procedure. The normal forms of C and "C $\sqcap$ D" are calculated with the procedure of normalisation.

There are two steps in the comparison. We compare the strict parts of the two concepts. If these are equal, then we compare the default parts. If the two normal forms are equal, the algorithm returns "Yes". It returns "No" otherwise.

The completeness, correctenness and the polynomial computation of $JClassic_{\delta\epsilon}$ have been proved in [4].

We detailed in the next section the connective "lcs".

IV. The computation of "lcs"

The least common subsumer has been introduced in description logic by Borgida et al. [5] as an external operation to compute the LCS of two concepts.

The LCS of two concepts A and B belonging to a language L is the most specific concept in L that subsumes both A and B.

Definition 3: Let L a terminological language, $\sqsubseteq$ the notation of subsumption relation in L

LCS: L $\times$ L $\rightarrow$ L

LCS(A,B) $\rightarrow$ C $\in$ L iff:

$A \sqsubseteq B$ and $B \sqsubseteq C$ (C subsume both A and B),

$\nexists$ D $\in$ L such that $A \sqsubseteq D, B \sqsubseteq D$ and $D \subsetneq C$ (i.e., there is no common subsumers to A and B which is subsumed strictly by C)

The next algorithm is to compute the LCS where input are the normal form of two concepts A_1 and A_2 and the output is the LCS of A_1 and A_2.

Let *a* and *b* two normal forms *A* and *B* with *a* and *b* $\neq b_0$ (b_0 is the normal form of $\perp$).

Algorithm 1 LCS

```
Require: a=$\prec a_\theta$,$a_\theta \succ$C and b=$\prec b_\theta$,$b_\theta \succ$C two normal forms of A and B.
Ensure: c=$\prec c_\theta$,$c_\theta \succ$C the normal form of LCS(A,B)
  $c_{\theta\pi} \leftarrow a_{\theta\pi} \cap b_{\theta\pi}$
  $c_{\theta r} \leftarrow \emptyset$
  for all $\prec$r, d $\succ \in$ a$_{\theta r}$ do
    if $\exists \prec$r, e $\succ \in$ b$_{\theta r}$ then
      f $\leftarrow$ LCS(d,e)
      $c_{\theta r} \leftarrow c_{\theta r} \cup \prec$r, f $\succ$
    end if
  end for
  $c_{\theta\epsilon} \leftarrow a_{\theta\epsilon} \cap b_{\theta\epsilon}$
  $c_{\delta\pi} \leftarrow a_{\delta\pi} \cap b_{\delta\pi}$
  $c_{\delta r} \leftarrow \emptyset$
  for all $\prec$r, d $\succ \in$ a$_{\delta r}$ do
    if $\exists \prec$r, e $\succ \in$ b$_{\delta r}$ then
      f $\leftarrow$ LCS(d,e)
      $c_{\delta r} \leftarrow c_{\delta r} \cup \prec$r, f $\succ$
    end if
  end for
  $c_{\delta\epsilon} \leftarrow a_{\delta\epsilon} \cap b_{\delta\epsilon}$
```

Our system can be used in differents application; we choose to use to model an access control model.

V. Application to Access Control

To show how we can use our description logic-based system and how we can infer a new knowledge, we define a knowledge base adapted to formalize a dynamic access control model.

In this model, authorization to subject are assigned depending on context. We consider first that the context is by default normal, and we represent it using the operator of default (δ). Then, each change of context is considered as an exception to the current context, this change is represented by the operator of exception (ϵ). We give, as an example, one ABox of a medical information system to show how authorization can be deduced.

- Using the instances in Table 1, the system infers that in organization **X**, each person who play the role of **Patient** is by default permitted to **consult** his **Med-rec** and add this instance to the ABox : $\delta Permission(P1)$.

where:

$\delta Permission(P1) \sqsubseteq PermisionAv.Activity(Consult) \sqcap PermissionR.Role(Patient) \sqcap PermissionV.View(Med-rec) \sqcap PermissionOr.Organization(X)$

Using the previous ABox, we show how deduction can be done in differents contexts.

- **Access control in a default context**: Suppose that user Marc wants to read Med-rec1; can he obtains that privilege?
  We know that:
  -Marc plays role of Patient in organization X: Employ(E1);
  -and, Med-rec1 is an object used in the view Med-rec: Use(U1);
  -and, Read is considered as a consultation activity: Consider(C1);
  -and finally, by default, in organization X, each person who plays the role of Patient is permitted to consult his Medical records, when Normal context is true: $\delta Permission(P1)$.
  Formally, we write:
  $Employ(E1) \sqcap Use(U1) \sqcap Consider(C1) \sqcap \delta Permission(P1)$
  Using security rules, we can deduce that the preceding proposition subsumes $\delta Is-permitted(I1)$.
  where:
  $Is-permitted(I1) \sqsubseteq Is-permittedAc.Action(Read) \sqcap Is-permittedS.Subject(Marc) \sqcap Is-pemittedO.Object(Med-rec1)$
  And because $Is-permitted(I1) \sqsubseteq \delta Is-permitted(I1)$, we can deduce that Marc is permitted to read his medical records.
- **Access control if context "Serious-disease" is true**: Suppose that Marc has a serious disease and he wants to read his medical records; did he have this right?
  In the context **Serious-disease**, the system deduce a new instance P2 and we add to the ABox the next rule:
  $Permision(P1)^\epsilon \sqsubseteq \delta Permission(P2)$
  We know that:
  -Marc plays role of Patient in organization X: Employ(E1);
  -and, Med-rec1 is an object used in the view Med-rec: Use(U1);
  -and, Read is considered as a consultation activity: Consider(C1);
  -and finally, by default, in organization X, each person who plays the role of Patient is permitted to consult his Medical records, when context Serious-disease is true: $\delta Permission(P2)$.
  We obtain:
  $Employ(E1) \sqcap Use(U1) \sqcap Consider(C1) \sqcap \delta Permission(P2)$
  $\equiv Employ(E1) \sqcap Use(U3) \sqcap Consider(C1) \sqcap \delta Permission(P1)^\epsilon$

ABox
Organization(X);
Role(Patient);
Subject(Marc);
View(Med-rec);
Object(Med-rec1);
Action(Read);
Activity(Consult);
$Employ(E1) \sqsubseteq EmployS.Subject(Marc) \sqcap EmployR.Role(Patient)$
$\sqcap EmployOr.Organization(X)$;
$Use(U1) \sqsubseteq UseO.Object(Med-rec1) \sqcap UseV.view(Med-rec)$
$\sqcap UseOr.Organization(X)$;
$Consider(C1) \sqsubseteq ConsiderAc.Action(Read) \sqcap ConsiderAv.Activity(Consult)$
$\sqcap ConsiderOr.Organization(X)$;

TABLE I
ABOX

We know that $A^\epsilon \equiv \delta A^\epsilon$, we obtain:
$\equiv Employ(E1) \sqcap Use(U1) \sqcap Consider(C1) \sqcap Permission(P1)^\epsilon$
Using security rules, we can deduce that the precedent proposition subsumes $Is-permitted(I1)^\epsilon$.
And, because $Is-permitted(I1) \not\sqsubseteq Is-permitted(I1)^\epsilon$, we cannot deduce Is-permitted(I1). Therefore Marc is not permitted to read his medical records when he has a serious disease.

Our policy language allows us to have more than one exception in a context. Exception at an even level cancel the effects of exceptions and therefore infers the property by default [6].

Supose that we have a disjunction of context, for example "context default or context serious disease", here we can use the connective "lcs" to deduce permission

- **lcs(default context, context of serious disease)**: Suppose that user Marc wants to read Med-rec1; can he obtain that privilege?
We know that:
-Marc plays role of Patient in organization X: Employ(E1);
-and, Med-rec1 is an object used in the view Med-rec: Use(U1);
-and, Read is considered as a consultation activity: Consider(C1);
and we have the two previous permissions Permission(P1) and Permission(P2), defined respectively for the default context and context of serious disease.
We obtain:
$Employ(E1) \sqcap Use(U1) \sqcap Consider(C1) \sqcap lcs(\delta Permission(P1), \delta Permission(P2))$
$\equiv Employ(E1) \sqcap Use(U3) \sqcap Consider(C1) \sqcap lcs(\delta Permission(P1), \delta Permission(P1)^\epsilon)$
using lcs properties, we obtain:
$\equiv Employ(E1) \sqcap Use(U1) \sqcap Consider(C1) \sqcap \delta Permission(P1)$
Using security rules, we can deduce that the precedent proposition subsumes $\delta Is-permitted(I1)$.
And, because $Is-permitted(I1) \sqsubseteq \delta Is-permitted(I1)$, we can deduce Is-permitted(I1). Therefore Marc is permitted to read his medical records when he has a serious disease or when the context is normal.

VI. CONCLUSION

The work presented in this paper has led to the definition of a new system based on description logic expressive enough to be used as part of an application and to represent default knowledge and exceptional knowledge. The $JClassic^+_{\delta\epsilon}$ highlights the interests and the relevance of defaults in conceptual definition. For the $JClassic^+_{\delta\epsilon}$ language, we have given a set of axioms outlining the essential properties of the connectives from this definitional point of view: property links default characteristics to exceptional or strict ones. This set of axioms induces a class of $CL^+_{\delta\epsilon}$-algebra of which the terms are concept descriptions. Using the conjunction connectives $\sqcap$ and "lcs", the set of concept can be partially ordered w.r.t the equational system (descriptive subsumption in free algebra). $JClassic^+_{\delta\epsilon}$ is defined with a universel algebraic corresponding to a denotational semantic, where terms are denoted exactly by sets of strict and default properties.

This system consists of three modules: a module for representing knowledge, a module to use that knowledge and a module to update knowledge. The module which allows to use knowledge is endowed with a subsumption algorithm which is correct, complete and polynomial.

In our work, the description logic is endowed with an algebraic intensional semantics, in which concepts are denoted by a normal form of all their properties. These normal forms (i.e., elements of the intensional semantic) are used directly as an input to the algorithm of subsumption and algorithm of deductive inferences.

To show how we can use our system, we applied it to access control. We developed a contextual access control model in which authorization are assigned to users depending on context. We represent this kind of authorization using the two operators of default (δ) and exception (ϵ).

An interseting topic for future research is to extend our tool to take into account spacial-temporal context to make our system more expressive with keeping a reasonable complexity. We also envisage to explore other appropriate and real applications.

REFERENCES

[1] V. Ventos and P. Brésellec. Least Common Subsumption as a connective. In *Proceeding of International Workshop on Description Logic*, Paris, France, 1997.

[2] V. Ventos, P. Brésellec and H. Soldano. Explicitly Using Default Knowledge in Concept Learning: An Extended Description Logics Plus Strict and Default Rules. In *Logic Programming and Nonmonotonic Reasoning, 6th International Conference, LPNMR 2001, pp. 173-185*, Vienna, Austria, September 17-19, 2001.

[3] F.M. Donini, M. Lenzerini, D. Nardi, and W. Nutt. The complexity of concept languages. In *Principles of language representation and reasoning: second international conference, pp. 151-162*, 1991.

[4] N. Boustia and A. Mokhtari. A dynamic access control model. In *Applied Intelligence Journal, DOI 10.1007/s10489-010-0254-z*, 2010, To appear.

[5] A. Borgida and P.F. Patel-Schneider Complete algorithm for subsumption in the CLASSIC description logic. *Artificial Intelligence Research, vol 1, pp. 278-308*, 1994.

[6] F. Coupey and C. Fouqueré. Extending conceptual definitions with default knowledge. *Computational Intelligence, vol 13, no 2, pp. 258-299* , 1997.

[7] N. Boustia and A. Mokhtari. Representation and reasoning on ORBAC: Description Logic with Defaults and Exceptions Approach. In *Workshop on Privacy and Security - Artificial Intelligence (PSAI), pp. 1008-1012, ARES'08, Spain*, 2008.

[8] N. Boustia and A. Mokhtari. $DL_{\delta\epsilon} - OrBAC$: Context based Access Control. In *WOSIS'09, pp. 111-118, Italy*, 2009.

[9] R.J. Brachman, D.L. McGuinness, P.F. Patel-Schneider, L. Alperin Resnick, and A. Borgida. Living with CLASSIC: When and How to Use a KL-ONE-Like Language. In *John Sowa, ed., Principles of Semantic Networks: Explorations in the representation of knowledge, pp. 401-456*, Morgan-Kaufmann: San Mateo, California, 1991.

[10] R.J. Brachman, D.L. McGuinness, L. Alperin Resnick, and A. Borgida. CLASSIC: A Structural Data Model for Objects. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, pp. 59-67*, June 1989.

A Tool for the Evaluation of the Complexity of Programs Using C++ Templates

Nicola Corriero, Emanuele Covino and Giovanni Pani
Dipartimento di Informatica
Università di Bari
Italy
Email: (corriero|covino|pani)@di.uniba.it

***Abstract*—We investigate the relationship between C++ template metaprogramming and computational complexity, showing how templates characterize the class of polynomial-time computable functions, by means of template recursion and specialization. Hence, standard C++ compilers can be used as a tool to certify polytime-bounded programs.**

***Keywords*-Partial evaluation; C++ template metaprogramming; polynomial-time programs.**

I. Introduction

According to [7], template metaprograms consist of classes of templates operating on numbers and types as a data. Algorithms are expressed using template recursion as a looping construct and template specialization as a conditional construct. Template recursion involves the use of class templates in the construction of its own member type or member constant. Templates were introduced to C++ to support generic programming and code reuse through parameterization. This is done by defining generic functions and objects whose behaviour is customized by means of parameters that must be known at compile time, entirely. For example, a generic vector class can be declared in C++ as follows:

```
template <class T, int N> class vector
  { T data[N]; };
```

The class has two parameters: T, the type of the vector's elements, and N, the length of the vector. The command line vector<int,5> instantiates the template by replacing all occurrences of T and N in the definition of vector with int and 5, respectively.

Templates are also able to perform static computation. The first example of this behaviour was reported in [22] and [23], where a program that forces the compiler to calculate (at compile time) a list of prime numbers is written; this ability is largely described by [7], [25], [26] and [9]: C++ may be regarded as a 2-level language, in which types are first-class values, and template instantiation mimics off-line partial evaluation. For instance, the following templates compute the function $pow(y, x) = x^y$;

```
template <int Y, int X> class pow
  {public: enum {result=X*pow<Y-1,X>::result };};

template <int X> class pow<0,X>
  {public: enum {result=1};};
```

The command line int z=pow<3,5>::result, produces at compile time the value 125, since the operator A::B refers to the symbol B in the scope of A; when reading the command pow<3,5>::result, the compiler triggers recursively the template for the values <2,5>, <1,5>, until it eventually hits <0,5>. This final case is handled by the partially specialized template pow<0,X>, that returns 1. Instructions like enum{result = function<args>::result;} represent the step of the recursive evaluation of function, and produce the intermediate values. This computation happens at compile time, since enumeration values are not l-values, and when one pass them to the recursive call of a template, no static memory is used (see [24], chapter 17). Thus, the compiler is used to compute *metafunctions*, that is as an interpreter for metaprogramming.

In [28], the following definition is given: a restricted metalanguage *captures* a property when every program written in the restricted metalanguage has the property and, conversely, for every unrestricted program with the property, there exists a functionally equivalent program written in the restricted metalanguage. An example of capturing a property by means of a restricted language is given in [3]: *any partial recursive function can be computed at compile-time* returning an error message that contains the result of the function. This is achieved specifying primitive recursion, composition, and μ-recursion by means of C++ template metaprogramming. A sketch of this result is in Section II-B.

On the other hand, the problem of defining syntactical characterizations of complexity classes of functions has been faced during the 90's; this approach has been dubbed *Implicit Computational Complexity* (ICC), and it aims at studying the complexity of programs without referring to a particular machine model and explicit bounds on time or memory. Several approaches have been explored for that purpose, like linear logic, rewriting systems, types and lambda-calculus, restrictions on primitive recursion. Two objectives of ICC are to find natural implicit characterizations of functions of various complexity classes, and to design systems suitable for static verification of programs complexity. In particular,

[2], [8], [11], [13], [14] studied how restricted functional languages can capture complexity classes.

In this paper we investigate the relationship between template metaprogramming and ICC, by defining a recursive metalanguage by means of C++ templates; we show that it captures the set of polynomial-time computable functions, that is, functions computable by a Turing machine in which the number of moves — the time complexity — is bounded by a polynomial. We also show that our approach can be extended to recursion schemes that are more general than those used in ICC. This result makes two contributions. First, it represents an approach to the automatic certification of upper bounds for time consumption of metaprograms. The compilation process certifies the complexity of the program, returning a specific error when the complexity is not polynomial. Second, there are few, if not any, characterizations of complexity classes made by metaprogramming; in particular, the result is achieved with a real, industrial template language, one that was constructed for doing real programming. Moreover, we do not define any extension of the language; we simply use the existing C++ type system to perform the computation.

The paper is organized as follows: in Section II we discuss some works related with our approach, and we recall some known results that we will use later; in Section III we show how to represent some polytime computable functions by means of template metaprogramming and how to rule out those functions that are not polytime (this is done by imposing some restrictions on the role of the template arguments); in Section IV we define the *Poly-Temp* language; in Section V we show that *Poly-Temp* is equivalent to the class of polynomial-time computable functions; finally, conclusions and further work are in Section VI.

II. Related works

A. C++ metaprogramming and functional programming

The prevailing style of programming in C++ is imperative. However, the mechanics of C++ metaprogramming shows a clear resemblance to dynamically-typed functional language, where all metaprograms are evaluated at compile time. This is clearly stated in [19]: they extended the purely functional language Haskell with compile-time metaprogramming (i.e., with a template system *à la* C++), with the main purpose to support the algorithmic construction of programs at compile-time.

In [12], it is recalled how function closure can be modelled in C++ by enclosing a function inside an object such that the local environment is captured by data members of the object; this idiom can be generalized to a type-safe framework of C++ class templates for higher-order functions that supports composition and partial application, showing that object-oriented and functional idioms can coexist productively. In [15] and [16], a rich library supporting functional programming in C++ is described, in which templates and C++ type inference are used to represent polymorphic functions. Another similar approach is in [20], where a functional language inside C++ is provided by means of templates and operator overloading. Both approaches provides functional-like libraries in run-time, while computations made by means of our metalanguage are performed at compile-time, totally. Coevally, [10] developed the template-implemented Lambda Library, which adds a form of lambda functions to C++. All these approaches lead to the introduction of lambda expression in C++ standard.

B. The computational power of C++ compilers

The first attempt to use C++ metaprogramming to capture a significant class of functions has been made by [3]; they presented a way to specify primitive recursion, composition, and μ-recursion by means of C++ templates. The result is not astonishing, provided that C++ templates are Turing complete (see [27]), but the reader should note that the technique used in the paper is based on the partial evaluation process performed by C++ compilers.

Number types are used to represent numbers; the number type representing zero is `class zero { }`. Given a number type `T`, the number type representing its unary successor is `template<class T> class suc { typedef T pre;}`.

A function is represented by a C++ class template, in which templates arguments are the arguments of the function. For example,

```
template<class T> class plus2
  {typedef suc<suc<T>> result;};
```

is a function type that computes the function $f(x) = x + 2$. The instructions

```
plus2<suc<zero>>::result tmp;
return (int) tmp;
```

returns an error message including `suc<suc<suc<zero>>>`, that is the value of $f(1)$. In particular, given a two-variable function f defined by primitive recursion from g and h,

$$\begin{cases} f(0,x) = & g(x) \\ f(y+1,x) = & h(y,x,f(y,x)) \end{cases}$$

function type `F` of f is expressed by the following templates, where `G` and `H` are the class templates computing g and h.

```
template <class Y, class X> class F
  {typedef typename
      H<typename Y::pre, X,
      typename F<typename Y::pre, X>::result >
                    ::result result;};

template <class X> class F<zero,X>
  {typedef typename G<X>::result result; };
```

Similar templates can be written to represent composition and μ-recursion, and to extend the definition to the general case on n variables; thus, the whole class of partial recursive functions can be expressed by template metaprogramming. For instance, templates times and pow can be defined as follows, where add returns the sum of its two arguments and is defined in the same way.

```
template<class Y, class X> class times
  {typedef typename
      add<X, typename times<typename Y::pre, X>
          ::result>::result result;};

template <class X> class times<zero,X>
  {typedef zero result};

template<class Y, class X> class pow
  {typedef typename
      times<X, typename pow<typename Y::pre,X>
                                    ::result>::result result;};

template <class X> class pow<zero,X>
  {typedef zero result}.
```

C. Capturing complexity classes by function-theoretic characterization

Syntactical characterizations of relevant classes of functions have been introduced by the Implicit Computational Complexity approach, that studies how restricted functional languages can capture complexity classes; in general, several restricted recursion schemes have been introduced, all sharing the same feature: no explicit bounds (as in [4]) are imposed in the definition of functions by recursion.

In order to show the relation between template metaprogramming and polynomial-time computable functions we need to recall that this class is defined in [2] as the smallest class B containing some initial functions, and closed under *safe recursion on notation* and *safe composition*. This result is obtained by imposing a syntactic restriction on variables used in the recursion and composition; they are distinguished in normal or safe, and the latter cannot be used as the principal variable of a function defined by recursion. In other words, one does not allow (safe) recursive terms to be substituted into a (normal) position which was used for an earlier definition by recursion. Normal inputs are written to the left, and they are separated from the safe inputs by means af a semicolon. A function in B can be written as $f(\vec{x};\vec{y})$; in this case, variables x_i are normal, whereas variables y_j are safe.

B is the smallest class of functions containing the initial functions 1-5 and closed under 6 and 7.

1) **Constant:** 0 (it is 0-ary function).
2) **Projection:** $\pi_j^{n,m}(x_1 \ldots, x_n; x_{n+1} \ldots, x_{n+m}) = x_j$, for $1 \le j \le m+n$.
3) **Binary successor:** $s_i(;a) = ai$, $i \in \{0,1\}$.
4) **Binary predecessor:** $p(;0) = 0$, $p(;ai) = a$.
5) **Conditional:**
$$C(;a,b,c) = \begin{cases} b & \text{if } a \bmod 2 = 0 \\ c & \text{otherwise.} \end{cases}$$
6) **Safe recursion on notation:** the function f is defined by safe recursion on notation from functions g and h_i if
$$\begin{cases} f(0,\vec{x};\vec{a}) = & g(\vec{x};\vec{a}) \\ f(yi,\vec{x};\vec{a}) = & h_i(y,\vec{x};\vec{a},f(y,\vec{x};\vec{a})). \end{cases}$$
for $i \in \{0,1\}$, g and h_i in B; y is called the *principal variable* of the recursion.
7) **Safe composition:** the function f is defined by safe composition from functions h, $\vec{r}$ and $\vec{t}$ if
$$f(\vec{x};\vec{a}) = h(\vec{r}(\vec{x};);\vec{t}(\vec{x};\vec{a}))$$
for h, $\vec{r}$ and $\vec{t}$ in B.

When defining a function $f(yi,\vec{x};\vec{a})$ by safe recursion on notation from g and h_i, the value $f(y,\vec{x};\vec{a})$ is in a safe position of h_i (right-side of the semicolon); and a function having safe variables cannot be substituted into a normal position of any other function, according to the definition of safe composition. Moreover, normal variables can be moved into a safe position, but not viceversa. By constraining recursion and composition in such a way, class B results to be equivalent to the class of polynomial time computable functions.

III. Template representations of some polytime functions

We show how to represent some polytime computable functions by means of template metaprogramming, imposing some restrictions on the role of the template arguments, following the mechanism introduced in [2]. Functions $\oplus$ and $\otimes$ can be expressed by safe recursion as follows:

$$\begin{cases} \oplus(0;x) = & x \\ \oplus(y+1;x) = & succ(;\oplus(y;x)). \end{cases}$$

$$\begin{cases} \otimes(0;a) = & 0 \\ \otimes(b+1;a) = & \oplus(a;\otimes(b;a)). \end{cases}$$

The recursive call $\otimes(b;a)$ is assigned to the safe variable x of $\oplus$, and one cannot re-assign this value to a normal variable of $\oplus$ (by definition 7, previous section, one cannot assign a function with safe variables to a normal position). For this reason, the following definition of $\otimes$ and both definitions of $\uparrow$ are not allowed in B.

$$\begin{cases} \otimes(0;a) = & 0 \\ \otimes(b+1;a) = & \oplus(\otimes(b;a);a). \end{cases}$$

$$\begin{cases} \uparrow(;0,x) = & 1 \\ \uparrow(;y+1,x) = & \otimes(x;\uparrow(;y,x)). \end{cases}$$

$$\begin{cases} \uparrow(;0,x) = & 1 \\ \uparrow(;y+1,x) = & \otimes(\uparrow(;y,x);x). \end{cases}$$

When defining the C++ templates that represent the previous three functions (or, in general, functions in B), we have to mimic the normal/safe behaviour by putting beside each variable a two-value flag; flags' values are defined according to the following rules:

1) each flag is equal to normal, initially;
2) flags beside variables assigned with recursive calls are changed to safe;
3) a compiler error must be generated whenever a variable labelled with a safe flag is used as principal variable of a recursion; this is done by adding a *negative specialization* (see below for its definition).

The template representation of $\oplus$ is the following (for sake of conciseness, we use enumeration values instead of typedef typename definitions, and integers instead of their class template representation):

```
♯define normal 0;
♯define safe 1;

template<int Y, int flagy, int X, int flagx> class sum
  {enum {result= 1+sum<Y-1, flagy, X, flagx>::result };};

template<int flagy, int X, int flagx>
  class sum<0, flagy, X, flagx> {enum {result= X };};

template<int Y, int X, int flagx>
  class sum<Y, safe, X, flagx>
  {enum {result= sum<Y, safe, X, flagx>::result };};
```

The instruction sum<2,normal,3,normal>::result returns the expected value, by recursively instantiating the first template sum for the values <2,3> and <1,3>, until <0,3> is reached (we omit here the flags); this value matches the second specialized template, which returns 3. The third template is introduced to avoid the substitution of other recursive calls or functions into variable Y, according to previous rule 3. This specialization is in the general form

```
template <args> class error <spec-args>
  {enum {result= error<spec-args>::result };};
```

and in this case the compiler stops, producing the error 'result' is not a member of type 'error<spec-args>'. The template representation of $\otimes$ is the following:

```
template<int Y, int flagy, int X, int flagx> class prod
  {enum {result=
  sum<X,flagx, prod<Y-1, flagy, X, flagx>::result, safe
          >::result}};

template<int flagy, int X, int flagx>
  class prod<0, flagy, X, flagx> {enum {result= 0};};

template<int Y, int X, int flagx>
  class prod<Y, safe, X, flagx>
  {enum {result= prod<Y, safe, X, flagx>::result}};
```

By rule 2, the flag associated with the recursive call of prod which occurs into sum is switched to safe, and by rule 3, the last template specialization is introduced to prevent the programmer from assigning another recursive call or function to Y. The instruction prod<2,normal,3,normal>::result instantiates the first template sum for values 3, normal, prod<1,normal,3,normal>::result, and safe, respectively; thus, the product is recursively evaluated. As shown above, one can also define prod by exchanging the arguments of sum, that is by assigning the recursive call of prod to the safe variable of sum, as follows:

```
template<int Y, int flagy, int X, int flagx> class prod
  {enum {result=
  sum< prod<Y-1, flagy, X, flagx>::result, safe, X, flagx
          >::result}};
```

The instruction prod<2,normal,3,normal>::result instantiates the template sum for values prod<1,normal, 3,normal>::result, safe, 3, and normal, respectively; this instantiation matches the values of the third template of sum's definition, and a compile-time error is produced (as expected, since we are trying to assign the recursive call of prod to the principal variable of sum). The template representation of the exponential function is

```
template<int Y, int flagy, int X, int flagx> class esp
  {enum {result=
  prod<esp<Y-1, flagy, X, flagx>::result, safe, X, flagx
          >::result}};

template<int flagy, int X, int flagx>
  class esp<0, flagy, X, flagx>
  {enum {result=1 };};

template<int Y, int X, int flagx>
  class esp<Y, safe, X, flagx>
  {enum {result= esp<Y, safe, X, flagx>::result}};
```

The instruction esp<2,normal,3,normal>::result instantiates the template prod for the values esp<1, normal, 3, normal>::result, safe, 3, and normal, respectively; this matches the third template of prod's definition, and a compiler error 'result' is not a member of type 'prod<1, safe, 3, normal>' is produced. If one exchanges the roles of prod's variables, the following template is written:

```
template<int Y, int flagy, int X, int flagx> class esp
  {enum {result=
  prod<X, flagx, esp<Y-1, flagy, X, flagx>::result, safe
          >::result}};
```

In this case the error occurs in the third template of sum's definition. In what follows, we will show that every partial recursive function in B can be represented by C++

templates that follow rules 1-3. If the specific error message is generated when compiling a function type F, this means that F represents a function f in a way that is not class B.

IV. Template representation of Polytime

In this section we define the Polytime language. Let normal and safe be notations for constants 0 and 1, respectively (this means, in C++ code, ♯define normal 0 and ♯define safe 1). *Binary number types* represent binary numbers, and are constructed recursively. We use the typedef typename mechanism (following [3]) instead of enumerated values; this allows us to write natural definitions of binary successors and predecessor and, in what follows, of composition and recursion on notation.

Number types representing the constant function 0 and binary successors of any number type T are in Figure 1; those representing the binary predecessor of any number type T are in Figure 2. According to these definitions, and intuitively using the composition template defined in Figure 4, the number 1101 can be represented by suc_1 <suc_0 <suc_1 < suc_1 <zero,safe>,safe>,safe>,safe>. The predecessor of any type number T is represented by pre<T, safe>::result. Each specialization has to implement the safe/normal behaviour on templates arguments (that is, on functions' variables). For example, it is mandatory in our system that the binary successors and the predecessor operate on safe arguments: thus, we add negative specializations to templates suc_0, suc_1 and pre, forcing them to produce a significant compiler error when the flag associated with the argument T is normal.

```
template<> class zero { typedef zero result;}

template<class T> class suc0 <T, safe>
       {typedef suc0 <T, safe> result;};

template<class T> class suc1 <T, safe>
       {typedef suc1 <T, safe> result;};

template<class T> class suc0 <T, normal>
       {typedef typename suc0 <T, normal>::result result;}

template<class T> class suc1 <T, normal>
       {typedef typename suc1 <X, normal>::result result;}
```

Figure 1: Templates for zero and binary successors

Templates for projection and conditional are defined in Figure 3. The first three specializations in myif definition are introduced to handle the cases in which the first argument C ends with 1 or 0, and the three arguments are safe, simultaneously. The fourth specialization returns an error when one or more arguments are normal.

The class template F that represents the *safe composition* of templates H, R and T is defined in Figure 4. Flags associated with R and T into H have values normal and safe, respectively; this implies that the value of T cannot be used by H as a principal variable of a recursion. The last specialization produces a compiler error if the variable X in R is safe, and R is used into H, simultaneously (X can be assigned with a safe value into R, harmlessly; but this cannot be done when R is substituted into a normal variable of H). This definition matches the definition of safe composition given in section II-C, and can be extended to the general case, when X and A are tuples of values, and R and T are tuples of templates.

```
template<> class pre<zero,safe> {typedef zero result;};

template<class T> class pre<suc0 <T, safe>, safe>
       {typedef T result;}

template<class T> class pre<suc1 <T, safe>, safe>
       {typedef T result;}

template<class T> class pre<suc0 <T, safe>, normal>
       {typedef typename pre<suc0 <T, safe>, normal>
              ::result result;}

template<class T> class pre<suc1 <T, safe>, normal>
       {typedef typename pre<suc1 <T, safe>, normal>
              ::result result;}
```

Figure 2: Templates for binary predecessor

```
template< class X1, int F1, . . ., class Xn, int Fn > class Πj
  {typedef Xj result }

template<class C, class X, class Y> class myif
  <suc1 <typename pre<C,safe>::result,safe>, safe,
          X, safe, Y, safe>
  {typedef Y result;};

template<class C, class X, class Y> class myif
  <suc0 <typename pre<C,safe>::result, safe>, safe,
          X, safe, Y, safe>
  {typedef X result;};

template<class C, class X, class Y>
  class myif <zero, safe, X, safe, Y, safe> {typedef X result;};

template<class C, int FC, class X, int FX, class Y, int FY >
  class myif
  {typedef typename if<C, FC, X, FX, Y, FY >
          :: result result;};
```

Figure 3: Templates for projections and conditional

We introduce now an extended definition of recursion, w.r.t. the definition given in [2]. A function f is defined by *n-ple safe recursion on notation* from functions h, $g_1 \dots g_n$, $m_1 \dots m_n$ if

$$\begin{cases} f(\vec{y}, \vec{x}; \vec{a}) = g_i(\vec{x}; \vec{a}) & \text{if one of } y_i \text{ is } 0 \\ f(\vec{y}, \vec{x}; \vec{a}) = h(\overrightarrow{m(y)}, \vec{x}; f(\overrightarrow{m(y)}, \vec{x}; \vec{a})) & \text{otherwise} \end{cases}$$

```
template<template<class X, int F_X > class R,
        class X, int F_X, class A, int F_A > class F
        {typedef typename
        H< typename R<X, F_X >::result, normal,
             typename T<X, F_X, A, F_A >::result, safe
             >::result result; };

template<template <class X> class R, class X, int F_X,
                            class A, int F_A >
   class F <R<class X, safe>, X, F_X, A, F_A >
   {typedef typename F<R<class X, safe>, X, F_X, A, F_A >
                            ::result result; };
```

Figure 4: Templates for safe composition

where $\vec{m(y)}$ stands for the sequence $m_1(y_1), \ldots, m_n(y_n)$, and each m_i is a sequence of binary predecessors.

Similarly, the class template F that represents the *n-ple safe recursion on notations* from templates H, G_1, ..., G_n and M_1, ..., M_n is defined in Figure 5, where each M_i ($i = 1 \ldots n$) is a sequence of predecessors applied to a binary number type which is not zero, and where we write template<X_1, F_1, ..., X_n, F_n > instead of template<class X_1, int F_1, ..., class X_n, int F_n >, for sake of simplicity. This definition can be extended to the general case, when X and A are tuples of variables.

```
template <Y_1, F_1, ..., Y_n, F_n, X, F_X, A, F_A > class F
  {typedef typename
  H<typename M_1 <Y_1 >::result, F_1,
       ...
       typename M_n <Y_n >::result, F_n,
       X, F_X, A, F_A,
       typename F<typename M_1 <Y_1 >::result, F_1,
                     ...
                     typename M_n <Y_n >::result, F_n,
                     X, F_X, A, F_A >::result,
       safe>::result result;};

template <Y_1, F_1, ..., Y_{i-1}, F_{i-1}, F_i,
             Y_{i+1}, F_{i+1}, ... Y_n, F_n, X, F_X, A, F_A >
  class F<Y_1, F_1, ..., Y_{i-1}, F_{i-1}, zero, F_i,
             Y_{i+1}, F_{i+1}, ... Y_n, F_n, X, F_X, A, F_A >
  {typedef typename G_i <X, F_X, A, F_A >::result result;};

template <Y_1, F_1, ..., Y_{i-1}, F_{i-1}, Y_i,
             Y_{i+1}, F_{i+1}, ... Y_n, F_n, X, F_X, A, F_A >
  class F<Y_1, F_1, ..., Y_{i-1}, F_{i-1}, Y_i, safe,
             Y_{i+1}, F_{i+1}, ... Y_n, F_n, X, F_X, A, F_A >
  {typedef typename F<Y_1,F_1,...,Y_{i-1},F_{i-1},Y_i,safe,
             Y_{i+1},F_{i+1},... Y_n,F_n,X,F_X,A,F_A >::result result;};
```

Figure 5: Templates for n-ple safe recursion

We set to safe the value of the flag associated with the recursive call of F into H (rule 2, Section III); we specialize F to compute the base cases of the recursion (where one of the templates G_i has to be computed); and we introduce the last n templates because the programmer is not allowed to assign a recursive call to one of the principal variables Y_1, ..., Y_n (rule 3).

We define the language *Poly-Temp* as the smallest class of templates containing zero, suc_0, suc_1, pre, myif, Π_j and closed under safe composition and n-ple safe recursion on notations. The polynomial-time functions will be represented exactly by those templates in *Poly-Temp* with all normal flags.

V. *Poly-Temp* CAPTURES POLYTIME

In this section, we state that every function computable within polynomial time by a Turing machine can be expressed in *Poly-Temp*; in order to do this, we recall that Polytime is captured by class B [2], and we prove that B is represented by templates in *Poly-Temp* (Theorem 5.1). Conversely, we show that any template in *Poly-Temp* is polynomial-space bounded (Theorem 5.2) and hence polynomial-time bounded (Theorem 5.3).

Theorem 5.1: For each function f in B, there exists a C++ template program F such that F computes f (at compile time).

Proof: (by induction on the construction of f). We denote binary number types with the capital letters X, Y, A, C, and the related flags with F_X, F_Y, F_A, F_C; we write (1) template<X, F_X, Y, F_Y > instead of template<class X, int F_X, class Y, int F_Y >; and (2) p<X> instead of typename pre<X,safe>::result, for sake of simplicity.

Base. Templates defined in section IV (constant, binary successors, predecessor, conditional and projections) trivially compute the basic functions of B.

Step. Case 1. Let f be defined by safe recursion on notations from functions $g(x; a)$, $h_0(y, x; a, s)$ and $h_1(y, x; a, s)$, that are computed, by the inductive hypotheses, by templates G, H_0 and H_1, respectively. f is represented in *Poly-Temp* by the following template F:

```
template <Y, F_Y, X, F_X, A, F_A > class F
{typedef typename myif<Y, safe
    typename H_0 <p<Y>, F_Y, X, F_X, A, F_A,
                    typename
                    F<p<Y>, F_Y, X, F_X, A, F_A >::result,
                    safe>::result, safe
    typename H_1 <p<Y>, F_Y, X, F_X, A, F_A,
                    typename
                    F<p<Y>, F_Y, X, F_X, A, F_A >::result,
                    safe>::result, safe
                        >::result result };

template <F_Y, X, F_X, A, F_A >
     class F<zero, F_Y, X, F_X, A, F_A >
     {typedef typename G<X, F_X, A, F_A >::result result};
```

```
template <Y, X, F_X, A, F_A >
    class F<Y, safe, X, F_X, A, F_A >
    {typedef typename F<Y, safe, X, F_X, A, F_A >
                        ::result result};
```

F is obtained by n-ple safe recursion (Figure 5) and safe composition (Figure 4) from templates if, H_0 and H_1.

Case 2. Let f be defined by safe composition from functions $h(p;q)$, $r(x;)$ and $t(x;a)$, that are computed, by the inductive hypotheses, by templates H, R and T, respectively. The template F computing f is defined in Figure 4. ■

To prove that any template in our language is polynomial-time bounded, we find a polynomial-space bound for the length of any template belonging to *Poly-Temp*. For sake of brevity, we omit the flags and we write safe inputs to the right of a semicolon, and normal ones to the left, following the notation used by Bellantoni and Cook. We also use "$(\ldots)$" instead of "$< \ldots >$". This implies that if a template F is defined by n-ple safe recursion from templates H, G_1, …, G_n and M_1, …, M_n, we write

$$\begin{aligned} \mathsf{F}(\overline{\mathsf{Y}}, \overline{\mathsf{X}}; \overline{\mathsf{A}}) &= \mathsf{G}_i(\overline{\mathsf{X}}; \overline{\mathsf{A}}) \quad \text{if one of } \mathsf{Y}_i \text{ is } \mathsf{zero} \\ &= \mathsf{H}(\overline{\mathsf{M(Y)}}, \overline{\mathsf{X}}; \overline{\mathsf{A}}, \mathsf{F}(\overline{\mathsf{M(Y)}}, \overline{\mathsf{X}}; \overline{\mathsf{A}})) \quad \text{otherwise} \end{aligned}$$

where $\overline{\mathsf{M(Y)}}$ stands for $\mathsf{M}_1(\mathsf{Y}_1), \ldots, \mathsf{M}_n(\mathsf{Y}_n)$, and each M_i is a sequence of binary predecessors.

Theorem 5.2: For each template F in *Poly-Temp*, there exists a polynomial q_F such that

$$|\mathsf{F}(\overline{\mathsf{X}}; \overline{\mathsf{A}})| \leq q_\mathsf{F}(\overline{|\mathsf{X}|}) + \max_i |\mathsf{A}_i|$$

where $\overline{\mathsf{X}}$ and $\overline{\mathsf{A}}$ are the variables labelled with normal and safe, respectively, and $q_\mathsf{F}(\overline{|\mathsf{X}|})$ stands for $q_\mathsf{F}(|\mathsf{X}_1|, \ldots, |\mathsf{X}_n|)$.

Proof: (by induction on the construction of F).

Base. If F is a constant, binary successors, predecessor, conditional or projection template, then we have $|\mathsf{F}(\overline{\mathsf{X}}; \overline{\mathsf{A}})| \leq 1 + \sum_i |\mathsf{X}_i| + \max_i |\mathsf{A}_i|$.

Step. Case 1. If F is defined by n-ple safe recursion we have, by induction hypotheses, the polynomials $q_{\mathsf{G}_1}, \ldots, q_{\mathsf{G}_n}$ and q_H bounding $\mathsf{G}_1, \ldots, \mathsf{G}_n$ and H, respectively; that is,

$|\mathsf{F}(\ldots, \mathsf{zero}, \ldots, \overline{\mathsf{X}}; \overline{\mathsf{A}})| \leq q_{G_j}(\overline{|\mathsf{X}|}) + \max_i |\mathsf{A}_i|$, and
$|\mathsf{F}(\overline{\mathsf{Y}}, \overline{\mathsf{X}}; \overline{\mathsf{A}})| \leq$
$q_H(\overline{|\mathsf{M(Y)}|}, \overline{|\mathsf{X}|}) + \max(\max_i |\mathsf{A}_i|, |\mathsf{F}(\overline{\mathsf{M(Y)}}, \overline{\mathsf{X}}; \overline{\mathsf{A}})|)$.

Define q_F such that

$$q_\mathsf{F}(\overline{|\mathsf{Y}|}, \overline{|\mathsf{X}|}) = \overline{|\mathsf{Y}|} \cdot q_\mathsf{H}(\overline{|\mathsf{Y}|}, \overline{|\mathsf{X}|}) + \textstyle\sum_j q_{\mathsf{G}_j}(\overline{|\mathsf{X}|}).$$

We have that $|\mathsf{F}(\ldots, \mathsf{zero}, \ldots, \overline{\mathsf{X}}; \overline{\mathsf{A}})| \leq q_\mathsf{F}(\overline{|\mathsf{zero}|}, \overline{|\mathsf{X}|}) + \max_i(\mathsf{A}_i)$. We also have

$$\begin{aligned} |\mathsf{F}(\overline{\mathsf{Y}}, \overline{\mathsf{X}}; \overline{\mathsf{A}})| &= |\mathsf{H}(\overline{\mathsf{M(Y)}}, \overline{\mathsf{X}}; \overline{\mathsf{A}}, \mathsf{F}(\overline{\mathsf{M(Y)}}, \overline{\mathsf{X}}; \overline{\mathsf{A}}))| \\ &\leq q_\mathsf{H}(\overline{|\mathsf{M(Y)}|}, \overline{|\mathsf{X}|}) + \\ &\qquad \max(\max_i |\mathsf{A}_i|, |\mathsf{F}(\overline{\mathsf{M(Y)}}, \overline{\mathsf{X}}; \overline{\mathsf{A}})|) \\ &\leq q_\mathsf{H}(\overline{|\mathsf{M(Y)}|}, \overline{|\mathsf{X}|}) + \\ &\qquad \max(\max_i |\mathsf{A}_i|, q_\mathsf{F}(\overline{|\mathsf{M(Y)}|}, \overline{|\mathsf{X}|}) + \max_i |\mathsf{A}_i|) \\ &\leq q_\mathsf{H}(\overline{|\mathsf{M(Y)}|}, \overline{|\mathsf{X}|}) + \\ &\qquad q_\mathsf{F}(\overline{|\mathsf{M(Y)}|}, \overline{|\mathsf{X}|}) + \max_i |\mathsf{A}_i| \\ &\leq q_\mathsf{H}(\overline{|\mathsf{M(Y)}|}, \overline{|\mathsf{X}|}) + \\ &\qquad \overline{|\mathsf{M(Y)}|} \cdot q_\mathsf{H}(\overline{|\mathsf{M(Y)}|}, \overline{|\mathsf{X}|}) + \textstyle\sum_j q_{\mathsf{G}_j}(\overline{|\mathsf{X}|}) + \\ &\qquad + \max_i |\mathsf{A}_i| \\ &\leq (\overline{|\mathsf{M(Y)}|} + 1) \cdot q_\mathsf{H}(\overline{|\mathsf{M(Y)}|}, \overline{|\mathsf{X}|}) + \\ &\qquad \textstyle\sum_j q_{\mathsf{G}_j}(\overline{|\mathsf{X}|}) + \max_i |\mathsf{A}_i| \\ &\leq \overline{|\mathsf{Y}|} \cdot q_\mathsf{H}(\overline{|\mathsf{M(Y)}|}, \overline{|\mathsf{X}|}) + \\ &\qquad \textstyle\sum_j q_{\mathsf{G}_j}(\overline{|\mathsf{X}|}) + \max_i |\mathsf{A}_i| \\ &\leq \overline{|\mathsf{Y}|} \cdot q_\mathsf{H}(\overline{|\mathsf{Y}|}, \overline{|\mathsf{X}|}) + \textstyle\sum_j q_{\mathsf{G}_j}(\overline{|\mathsf{X}|}) + \max_i |\mathsf{A}_i| \\ &\leq q_\mathsf{F}(\overline{|\mathsf{Y}|}, \overline{|\mathsf{X}|}) + \max_i |\mathsf{A}_i| \end{aligned}$$

Case 2. If f is defined by safe composition we have, by induction hypotheses, q_H, q_R and q_T bounding H, R and T, respectively; we have

$$\begin{aligned} |\mathsf{F}(\overline{\mathsf{X}}; \overline{\mathsf{Y}})| &= |\mathsf{H}(\mathsf{R}(\overline{\mathsf{X}};); \mathsf{T}(\overline{\mathsf{X}}; \overline{\mathsf{Y}}))| \\ &\leq q_\mathsf{H}(|\mathsf{R}(\overline{\mathsf{X}};)|) + |\mathsf{T}(\overline{\mathsf{X}}; \overline{\mathsf{Y}})| \\ &\leq q_\mathsf{H}(q_\mathsf{R}(\overline{|\mathsf{X}|})) + |\mathsf{T}(\overline{\mathsf{X}}; \overline{\mathsf{Y}})| \\ &\leq q_\mathsf{H}(q_\mathsf{R}(\overline{|\mathsf{X}|})) + q_\mathsf{T}(\overline{|\mathsf{X}|}) + \max_j |\mathsf{Y}_j| \end{aligned}$$

Let $q_\mathsf{F}(\overline{|\mathsf{X}|}, \overline{|\mathsf{Y}|})$ be $q_\mathsf{H}(q_\mathsf{R}(\overline{|\mathsf{X}|})) + q_\mathsf{T}(\overline{|\mathsf{X}|})$. We have the result. ■

Note that templates in *Poly-Temp* are polynomially time-bounded too, when evaluated. Indeed, base templates (zero, Π_j, suc_0, suc_1, if) are bounded by the length of their arguments; for composition templates, observe that the composition of two polynomial-time templates is still a polynomial time template; for recursion templates, it is well known that recursion on notation can be executed in polynomial time if the result of the recursion is polynomially length-bounded and the step and base functions are polytime, as in our case. Thus, we have

Theorem 5.3: Each template F in *Poly-Temp* is evaluated in polynomial time.

VI. Conclusions and further work

In summary, we have defined a restricted metalanguage by means of C++ templates, and we have shown that it captures at compile time the set of polynomial-time computable functions. As we mentioned in the Introduction, a contribution of this result is that it could provide the theoretical base for the construction of tools for the formal certification of upper bounds for metaprogramming time consumption. As an anonymous referee says, "normal" meta-programs do not follow the restriction imposed by our metalanguage; thus, a sensible prosecution of this work could be the analysis of transformation methods from "normal" metaprograms to "restricted" ones. Nevertheless, even if our template language is admissible C++, there is no doubt that programming in it should be hard, due to the extra annotations encoded as templates parameters; one may think to hide them into *traits* [18] containing representation of numbers and of related flags; in this way we'd be able to obtain a neater language. However, obscure error messages from C++ compilers could inhibit this as a workable approach. The programmer is not able to understand why and where in the program he used a recursive variable in the wrong way; *static interfaces* techniques [17] could help us to provide a clearer meaning to error messages.

Even if this is a clumsy characterization of a complexity class, it is worth noting that the three rules introduced above can produce polynomial time bounded templates when applied to all kinds of recursions, not only to primitive recursion; it seems that our approach improves the understanding of polynomial-time computation's nature, allowing us to use more expressive recursive schemes.

References

[1] D. Abrahams and A. Gurtovoy, *C++ template metaprogramming: concepts, tools and techniques from Boost and beyond*, Addison Wesley, 2004.

[2] S. Bellantoni and S. Cook, "A new recursion-theoretic characterization of the poly-time functions", *Computational Complexity*, vol. 2, pp. 97–110, 1992.

[3] M. Böhme and B. Manthey, "The computational power of compiling C++", *Bull. of the EATCS*, vol. 81, pp. 264–270, 2003.

[4] A. Cobham, "The intrinsic computational difficulty of functions", in *Y. Bar-Hillel (ed), Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science, North-Holland, Amsterdam*, 1962, pp. 24–30.

[5] L. Colson, " About primitive recursive algorithms", *Theoretical Computer Science*, vol. 83, pp. 57–69, 1991.

[6] L. Colson and D. Fredholm, " System T, call by value and the minimum problem", *Theoretical Computer Science*, vol. 206, pp. 301–315, 1998.

[7] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

[8] M. Hofmann, " Linear type and non-size-increasing polynomial time computation", in *Proceedings of the 14th IEEE Symposium on Logic in Computer Science*, 1999, pp. 464–473.

[9] J. Järvi, " Compile Time Recursive Objects in C++", in *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS '98), Washington DC, USA*, 1998.

[10] J. Järvi, G. Powell, and A. Lumsdaine, " The Lambda library: unnamed functions in C++", *Softw. Pract. Exper.*, vol. 33, pp. 259–291, 2003.

[11] N.D. Jones, " LOGSPACE and PTIME characterized by programming languages", *Theoretical Computer Science*, vol. 228, pp. 151–174, 1999.

[12] K. Läufer, " A Framework for Higher-Order Functions in C++", in *Proceedings of the Conf. on Object-Oriented Technologies (COOTS), Monterey, CA*, 1995.

[13] D. Leivant, "Ramified recurrence and computational complexity I: word recurrence and polytime", in P. Clote, J. Remmel (eds), Feasible Mathematics II, Birkauser, 1994.

[14] D. Leivant and J.-Y. Marion, "Ramified recurrence and computational complexity II: substitution and polyspace", in J. Tiuryn, L. Pocholsky (eds), Computer Science Logic, LNCS, vol. 933, pp. 486–500, 1995.

[15] B. McNamara and Y. Smaragdakis, " Functional programming in C++", in *Proceedings of the 5th ACM SIGPLAN International conference on Functional programming, ICFP '00, New York NY, USA*, 2000, pp. 118–129.

[16] B. McNamara and Y. Smaragdakis, "Functional programming with the FC++ library", *J. Funct. Program.*, vol. 14(4), pp. 429–472, 2004.

[17] B. McNamara and Y. Smaragdakis, "Static Interfaces in C++", *First Workshop on C++ Template Programming, Erfurt, Germany*, 2000.

[18] N. Myers, " A new and useful template technique: "Traits" ", *C++ Report*, vol. 7(5), pp. 32–35, 1995.

[19] T. Sheard and S. Peyton Jones, " Template meta-programming for Haskell", in *Proceedings of the 2002 Haskell Workshop (Haskell '02), Pittsburgh, PA*, 2002, pp. 1–16.

[20] J. Striegnitz, " The FACT! library homepage", 2000. Available (September 2011): http://www.fz-juelich.de/zam/FACT

[21] B. Stroustrup, *The C++ programming language*, Addison-Wesley, 1997.

[22] E. Unruh, "Prime number computation, ANSI X3J16-94-0075/ISO WG21-462".

[23] E. Unruh, " Template metaprogrammierung", 2002. Available (September 2011): http://www.erwin-unruh.de/ meta.html

[24] D. Vandevoorde and N.M. Josuttis, *C++ templates: the complete guide*, Addison-Wesley, 2003.

[25] T. Veldhuizen, " Using C++ templates metaprograms", *C++ Report*, vol. 7(4), pp. 36–43,1995 .

[26] T. Veldhuizen, " C++ templates as partial evaluation", in *Proceedings of the 1999 ACM SIGPLAN Symposium on Partial Evaluation and Semantic-Based Program Manipulation*, 1999, pp. 13–18.

[27] T. Veldhuizen, " C++ templates are Turing complete", *unpublished.*

[28] T. Veldhuizen, " Tradeoffs in metaprogramming", in *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantic-Based Program Manipulation*, 2006, pp. 150–159.