# Completeness of the Owicki-Gries System for Parameterized Parallel Programs

Leonor Prensa-Nieto
Technische Universität München
Institut für Informatik, 80290 München, Germany
prensani@in.tum.de

## Abstract

*The Owicki-Gries system is extended to allow the verification of parameterized programs of the form* $\|_{i=1}^{n} S(i,n)$ *where for each* $n$ *we obtain a parallel program consisting of* $n$ *sequential processes. By the completeness of the original Owicki-Gries system, we know that for any valid specification of a parallel program with a fixed number of components there exists a proof in the system. However, we would like to know if there exists a single proof that works for any number of components. In this paper we prove completeness of the extended system, thus giving a positive answer to the question.*

## 1 Introduction

Susan Owicki and David Gries introduced a Hoare-like proof system for proving partial correctness of concurrent imperative programs with shared variables [6, 7, 8]. The system was proved to be sound and (relatively) complete with respect to the standard interpretation in the natural numbers. The proofs for parallel programs had to be given in the form of *proof outlines*, i.e., annotated at every control point where interferences with other components may occur. The proof rule for parallel composition was[1]

$$\frac{\begin{array}{c}\vdash \{P_1\}S_1\{Q_1\},\ldots,\vdash \{P_k\}S_k\{Q_k\}\\ \textit{and the proof outlines are interference free}\end{array}}{\vdash \{P_1\cap\ldots\cap P_k\}\ S_1\|\ldots\|S_k\ \{Q_1\cap\ldots\cap Q_k\}}$$

This rule is concerned with the verification of parallel programs consisting of a fixed number $k$ of (possibly different) components $S_1,\ldots,S_k$. We refer to the Owicki-Gries system by $O$.

[1] We model assertions as sets of states.

$$\begin{array}{ll} S(i,n)\equiv & \textbf{while}\ \textit{true}\ \textbf{do}\\ & \quad NCS(i,n);\ \ (*\text{noncritical section}*)\\ & \quad \langle\ \textit{turn}[i] := \textit{num};\ \textit{num} := \textit{num}+1\ \rangle;\\ & \quad \textbf{wait}\ \textit{turn}[i] = \textit{next}\ \textbf{end};\\ & \quad CS(i,n);\quad (*\text{critical section}*)\\ & \quad \textit{next} := \textit{next}+1\\ & \textbf{od}\end{array}$$

**Figure 1. Ticket algorithm**

Many interesting parallel programs, however, are often given schematically in terms of a parameter $n$, representing the number of parallel components. A typical example is the ticket mutual exclusion algorithm for distributed processes[2] [3]

$$\textbf{ticket} \equiv \textit{num} := 1;\ \textit{next} := 1;\ \textit{turn} := 0;\ \|_{i=1}^{n} S(i,n)$$

where $S(i,n)$ is shown in Figure 1. In order to be able to verify such programs, we extend the programming language with a new construct: $\|_{i=1}^{n} S(i,n)$, where $n$ ranges over the natural numbers and $S$ is a sequential program parameterized by $n$ and $i$ such that $1\le i\le n$. The syntax "$\|_{i=1}^{n} S(i,n)$" represents the program $S(1,n)\|\ldots\|S(n,n)$, where $\|$ is the parallel composition construct based on interleaving.

We extend the system $O$ with the following rule:

$$\frac{\begin{array}{l}\forall i.\ 1\le i\le n \longrightarrow \vdash \{P(i,n)\}S(i,n)\{Q(i,n)\}\ \wedge\\ \forall i,j.\ 1\le i,j\le n\wedge i\ne j \longrightarrow \textit{the proof outlines of}\\ \{P(i,n)\}S(i,n)\{Q(i,n)\}\wedge\{P(j,n)\}S(j,n)\{Q(j,n)\}\\ \textit{are interference free}\end{array}}{\vdash \{\bigcap_{i=1}^{n} P(i,n)\}\ \|_{i=1}^{n} S(i,n)\ \{\bigcap_{i=1}^{n} Q(i,n)\}}$$

where $P(i,n)$ and $Q(i,n)$ denote for each $i$ and $n$ the precondition and the postcondition, respectively.

[2] By the assignment $\textit{turn} := 0$ we mean that 0 is assigned to each component of the array $\textit{turn}$.

With this rule we can prove partial correctness of a parallel program parameterized by the number of component processes $n$ by showing it for an arbitrarily chosen but fixed value of $n$. We refer to the system of rules that results by adding this new rule to the Owicki-Gries system by $F$ ($F$ stands for *family*). We write $\vdash_F \{P\}S\{Q\}$ to denote that $\{P\}S\{Q\}$ can be derived in $F$. The soundness of this system has been already proven in [5]. In the current paper we will prove the relative completeness of $F$.

The proof proceeds by induction on the structure of the programs. The only interesting case here is that of parameterized parallel programs, i.e., given a family of valid partial correctness formulas of the form

$$\{P(n)\}\|_{i=1}^{n} S(i,n)\{Q(n)\}$$

we want to study if we can always find a proof outline for $S$ so that

$$\vdash_F \{P(n)\}\|_{i=1}^{n} S(i,n)\{Q(n)\}$$

By the completeness of the system $O$ we know that for each value of $n$ we can always find proof outlines for each of the $n$ components, so that the program can be derived in $O$. This result, however, does not give any information about what these proof outlines look like for different values of $n$ and $i$. Do they have a uniform pattern (probably parameterized by $n$ and $i$) or are the proof outlines of the program with three components completely different from those of the program with four components? And, for a given value of $n$, is the annotation of the component $i$ different from the annotation of the component $j$ for some values of $i$ and $j$ in the set $\{1,\ldots,n\}$?

Previous works on parallel program verification present proofs of parameterized programs in the style of Owicki and Gries [3]. However, they abstract from the fact that the Owicki-Gries system does not directly support reasoning on parameterized programs. An extension of the system and a proof of completeness was missing. The authors of [3] say

> Since Szymanski's algorithm[3] is parameterized, it is quite natural to use assertion networks parameterized by process index $i$.

In this paper we provide a formal proof that supports this "naturality". The main conclusion is that, for any valid specification of a parameterized program, it is always possible to find a single (parameterized) proof outline that can be derived in the system $F$ for all values of $n$.

[3] A mutual exclusion algorithm for distributed processes.

The paper is organized as follows. In §2 we formally define the class of parameterized programs considered for verification in our system. In §3 we prove the semantic existence of the intermediate assertions that form a proof outline. This means that we show in our meta-language (English and mathematics) that these assertions exist. Further, we prove that the so constructed proof outlines yield valid $F$-proofs (in a semantic way) since they satisfy the expected properties for using the method. In §4 we show that these (semantic) assertions are also expressable in a language containing at least first order arithmetic over the standard model of the natural numbers (elementary arithmetic). That is, we prove that syntactic expressions corresponding to the semantic ones exist. This result is called *expressiveness*, which together with the semantic completeness implies *relative completeness*.

## 2 Parameterized Programs

Parameterized programs represent a family of programs by a single syntactic object. In order to represent, understand and manipulate these objects, several aspects of the programming language have to be adapted to deal with parameters. In this section we give a formal description of the syntax of parameterized programs, their operational semantics and the meaning of Hoare logic for this class of programs.

### 2.1 Syntax of Parameterized Programs

Let $n$ and $i$ be parameters, where $n$ ranges over the natural numbers and $i$ ranges over the subset $\{1,\ldots,n\}$. A *parameterized component program* $S(i,n)$, parameterized by $n$ and $i$, is described by the following syntactic sets:

1. A finite set of simple variables $V = \{x_1,\ldots,x_k\}$, where each $x_j$, $1 \le j \le k$ ranges over the natural numbers, and a finite set of composed variables $V_c = \{y_1,\ldots,y_l\}$. Composed variables are usually implemented as arrays, where the value of an array is a function from some "range" to some "set" of values [2]. For example in the ticket algorithm the composed variable *turn* can be interpreted semantically as a function from the subset $\{1,\ldots,n\}$ to $\mathbb{N}$, then *turn*$[i]$ would represent the result of applying *turn* to $i$. We interpret composed variables directly on $\mathbb{N}$ in a way that we explain in §2.2.

2. Parameterized arithmetic and boolean expresions

given by the following BNF grammars:

$$
\begin{aligned}
a ::= &\ j \mid m \mid N \mid x \mid y[a_0] \mid a_0 + a_1 \mid a_0 - a_1 \mid \\
&\ a_0 * a_1 \\
b ::= &\ \mathbf{true} \mid b_0 \wedge b_1 \mid \neg b_0 \mid \exists x.\ b_0 \mid a_0 \leq a_1
\end{aligned}
$$

where $j$ and $m$ are special variables that take the values given by the parameters $i$ and $n$, $N$ is a natural number, $x \in V$ and $y \in V_c$.

Given an arithmetic expression $a$, we define $a(i,n)$ inductively as follows:

$$
\begin{aligned}
&j(i,n) \stackrel{\text{def}}{=} i, \quad m(i,n) \stackrel{\text{def}}{=} n, \quad N(i,n) \stackrel{\text{def}}{=} N, \\
&x(i,n) \stackrel{\text{def}}{=} x, \quad y[a_0](i,n) \stackrel{\text{def}}{=} y[a_0(i,n)], \\
&(a_0 + a_1)(i,n) \stackrel{\text{def}}{=} a_0(i,n) + a_1(i,n), \\
&(a_0 - a_1)(i,n) \stackrel{\text{def}}{=} a_0(i,n) - a_1(i,n), \\
&(a_0 * a_1)(i,n) \stackrel{\text{def}}{=} a_0(i,n) * a_1(i,n)
\end{aligned}
$$

For a boolean expression $b$, $b(i,n)$ is defined analogously.

3. Finally, the syntax of parameterized component programs:

$$
\begin{aligned}
S ::= &\ x := a \mid y[a_0] := a_1 \mid S_1;\ S_2 \mid \\
&\ \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi} \mid \\
&\ \mathbf{while}\ b\ \mathbf{do}\ S\ \mathbf{od} \mid \mathbf{await}\ b\ \mathbf{then}\ T\ \mathbf{end}
\end{aligned}
$$

We define $S(i,n)$ inductively as follows:

$$
\begin{aligned}
&(x := a)(i,n) \stackrel{\text{def}}{=} x := a(i,n), \\
&(y[a_0] := a_1)(i,n) \stackrel{\text{def}}{=} y[a_0(i,n)] := a_1(i,n), \\
&(S_1;\ S_2)(i,n) \stackrel{\text{def}}{=} S_1(i,n);\ S_2(i,n), \\
&(\mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi})(i,n) \stackrel{\text{def}}{=} \\
&\mathbf{if}\ b(i,n)\ \mathbf{then}\ S_1(i,n)\ \mathbf{else}\ S_2(i,n)\ \mathbf{fi} \\
&\ldots
\end{aligned}
$$

The synchronization construct **await**, whose body $T$ is a sequential program containing neither **await** nor **while** (to ensure termination), has the following semantics: when the boolean condition evaluates to true, the body is executed as an atomic operation, i.e., without interference from other processes. The notation $\langle S \rangle$ stands for **await** *true* **then** $S$ **end**, and **wait** $b$ **end** stands for **await** $b$ **then** *skip* **end**. Single assignments are also executed atomically.

The family of parallel programs built form $S(i,n)$ is defined by the set $\{\|_{i=1}^{n} S(i,n) \mid n \in I\!N\}$, where for each $n$, the resulting member is a concrete (non-parameterized) parallel program like the programs considered in the original Owicki-Gries system.

In the current paper parallel programs are of the form $S_1 \| \ldots \| S_k$, where $S_1, \ldots, S_k$ are sequential programs or parameterized parallel programs $\|_{i=1}^{n} S(i,n)$. That is, parallelism only occurs at the top level. However, they can be sequentially composed with other parallel programs (or with sequential programs seen as a degenerated case of parallel programs), and they can appear in if- and in while-constructions.

**Note:** The rule for parallel parameterized programs can be generalized to a rule where the parallel composition of several different parameterized programs can be derived. The conclusion then would be:

$$
\begin{array}{c}
\{\bigcap_{i=1}^{n_1} P_1(i,n_1)\ \cap \ldots \cap\ \bigcap_{i=1}^{n_k} P_k(i,n_k)\} \\
\|_{i=1}^{n_1} S_1(i,n_1) \ \| \ldots \| \ \|_{i=1}^{n_k} S_k(i,n_k) \\
\{\bigcap_{i=1}^{n_1} Q_1(i,n_1)\ \cap \ldots \cap\ \bigcap_{i=1}^{n_k} Q_k(i,n_k)\}
\end{array}
$$

where $n_1, \ldots, n_k \in I\!N$. The original Owicki-Gries rule can be obtained as an instance of this more general rule by just considering all $n_i$ to be one.

### 2.2 Semantics of Parameterized Programs

The meaning of a program $S$ is usually defined as a partial function $\mathcal{M}(S)$ from states to states. States are represented as tuples of values corresponding to the variables occurring in the program. For the purpose of our completeness proof it is convenient to require that these values be natural numbers. Thus, given a state $s$ and a simple variable $x$, we will refer to the value of the variable $x$ in state $s$ by $s(x)$. For a composed variable $y$, $s(y)$ returns the corresponding encoding of the sequence of values given by its components. This means that the state space of a composed variable is $I\!N$. As we will see in §4, the reason for this is that free variables in the language of elementary arithmetic must range over numbers and not over functions.

There are many different ways of encoding a finite sequence of values as a single natural number. Any such encoding will do as long as there is an effective procedure for encoding and an effective procedure for extracting any of the original components from the coded sequence. The encoding of the sequence $a_1, \ldots, a_k$ will be represented by $\lceil a_1, \ldots, a_k \rceil$. If $a = \lceil a_1, \ldots, a_k \rceil$ then $a \frown c = \lceil a_1, \ldots, a_k, c \rceil$. We denote the code of the empty sequence by $\lceil\rceil$. We will assume some properties of the functions "$\lceil \ldots \rceil$" and "$\frown$", in particular, their definability in the language of elementary arithmetic. The proofs of these properties can be found in [11].

The state space $\Sigma$ of a given parameterized program is then $I\!N^k$, where $k$ is the total number of variables appearing in the program. Having defined the state we can now define the operational semantics.

Parameterized programs are only a schematic representation of an infinite family of uniformly defined concrete programs. When we speak of execution of a parameterized program, we implicitly refer to the execution of any of the concrete programs it represents. Thus, the operational semantics for parameterized programs is given in terms of the semantics for an arbitrary concrete program, that is, for the parallel program given by an arbitrary but fixed value of the parameter $n$. A description of the rules defining the (interleaving) semantics of (non-parameterized) parallel programs can be found in [1]. Due to our special treatment of assignments to composed variables we briefly describe its semantics.

The operational semantics is defined by a set of transition rules between configurations, where a configuration is a pair of a program fragment and a state. A program fragment is either a program, or, if execution has come to an end, the empty program $E$. The rule for assignments to components of a composed variable is:

$$\langle y[a_0] := a_1, s\rangle \rightarrow \langle E, s[y \leftarrow \mathit{update}(y, s(a_0), s(a_1))]\rangle$$

The notation $s[y \leftarrow t]$ stands for the usual operation of substitution in $s$ of the value of the variable $y$ by the new value $t$. The syntax $s(a_0)$ denotes the value resulting from the evaluation of the arithmetic expression $a_0$ by substituting the free variables appearing in it by the values they have in the state $s$. And finally, $\mathit{update}(y, j, l)$ is a function that returns the encoding of the sequence that results from substituting in the sequence encoded by $y$ the $j$th component by the new value $l$. It is possible to prove that there is a partial recursive procedure computing this function. Thus, the transformations performed on the state by a program are all computable.

### 2.3 Hoare Logic for Parameterized Programs

A Hoare-style verification calculus relates imperative programs with assertions. Assertions are usually interpreted as first-order predicates on states, where free variables denote the values of program variables in a specific state.

For sequential programs, two assertions, namely, a precondition and a postcondition, were enough to develop a sound and (relatively) complete system of rules. In the Owicki-Gries system, however, assertions are needed at every control point where interferences with other concurrent processes are allowed.

Given a parameterized component program $S(i, n)$, we construct its proof outline associating an assertion with every point of interference. For this purpose, it suffices to annotate $S(i, n)$ with a postcondition $q(i, n)$ and every normal subprogram $R(i, n)$ of $S(i, n)$ with a precondition $\mathit{pre}(R(i, n))$, where a *normal subprogram* of a program $S$ is a subprogram of $S$ not occurring within any atomic region of $S$. The result will be an annotated program $A(S(i, n)))$. We define $\mathit{pre}(A(S(i, n)))$ and $\mathit{post}(A(S(i, n)))$ to be the initial precondition and the final postcondition, respectively.

**Definition 2.1 (Semantics of proof outlines for parameterized component programs)** *A proof outline $A(S(i, n))$ of a program $S(i, n)$ is valid (in the sense of partial correctness), written $\models_F A(S(i, n))$, iff for all $n$ and for all $i (1 \leq i \leq n)$ whenever the control of $S(i, n)$ in a given computation starting in a state $s \in \mathit{pre}(A(S(i, n)))$ reaches a point annotated by an assertion, this asssertion is true. In particular, if the computation terminates in a state $s'$, then $s' \in \mathit{post}(A(S(i, n)))$.*

**Definition 2.2 (Semantics of Hoare triples for parameterized parallel programs)** *A parameterized Hoare triple $\{P(n)\}\|_{i=1}^{n} S(i, n)\{Q(n)\}$ is valid (in the sense of partial correctness), and we write $\models_F \{P(n)\}\|_{i=1}^{n} S(i, n)\{Q(n)\}$, if and only if $\forall n.\ \models \{P(n)\}\|_{i=1}^{n} S(i, n)\{Q(n)\}$.*

$\models \{P\}S\{Q\}$ is the usual definition of validity for partial correctness for non-parameterized programs, i.e., every terminating computation of $S$ that starts in a state satisfying $P$ ends in a state satisfying $Q$.

## 3 Semantic Completeness

In this section we prove semantic completeness of the system $F$. The proof proceeds by induction on the structure of the programs. The case for parallel programs of the form $S_1\| \ldots \|S_k$ where the number of components is fixed was first proven by Owicki [6, 1].

The remaining case is that of parameterized parallel programs. Given a valid partial correctness formula of the form

$$\models_F \{P(n)\}\|_{i=1}^{n} S(i, n)\{Q(n)\}$$

we prove that we can always find a proof outline for $S$ so that

$$\vdash_F \{P(n)\}\|_{i=1}^{n} S(i, n)\{Q(n)\}$$

To prove correctness of parameterized parallel programs using the system $F$, we need to show local correctness and interference freedom of the associated proof outlines. To this end, one must in general augment actions and boolean guards with assignments to auxiliary variables. These record the progress of computation in the other processes allowing to express the effects of parallel execution.

There are two well-known canonical ways of introducing auxiliary variables. One makes use of a single so-called *history* variable $z$ that records, throughout execution, the index number of the component performing the atomic action and the values of the program variables before that action is executed. In a terminating state, $z$ contains the history associated with the executions of assignments or **await**-statements. The second posibility consists of using auxiliary variables as labels indicating the control points that the program passes through. Thus, these special auxiliary variables (which are also called counters) are updated with *every* transition. One auxiliary variable of this form for each component suffices for any Owicki-Gries style proof [2, 4].

Our programming language does not allow for the atomic evaluation of a boolean condition together with the updating of an auxiliary variable. As a consequence, not all transitions but only assignments and **await**-statements, can be recorded by auxiliary variables. For this reason we use a single history variable $z$, respecting this syntactic restriction, in the style of Apt [1]. The set of program variables is augmented with a single variable $z$, containing the value corresponding to the encoding of the substring of the history sequence, where transitions corresponding to the evaluation of boolean expressions are not registered.

Given a program $S$ with state space $\Sigma$, we denote the state space of the extended program $S^*$ by $\Sigma^*$, where we assume that the first value of each tuple corresponds to the value of the history variable. For instance, if $s^* \in \Sigma^*$ is a state of $S^*$, then, by convention, $s$ denotes the corresponding state of $\Sigma$, where the first element of the tuple has been removed.

Given $\models_F \{P(n)\}\|_{i=1}^{n} S(i,n)\{Q(n)\}$, the proof of semantic completeness is structured as follows:

1. Extend the program by initializing the history variable:
$$z := \lceil\rceil; \|_{i=1}^{n} S(i,n)$$
2. Extend $S(i,n)$ to $S^*(i,n)$ by adding assignments to the history variable $z$.
3. Annotate every point of interference in $S^*(i,n)$ with an assertion, obtaining an annotated program $A(S^*(i,n))$ such that the following conditions hold:
   (a) $\forall s \in \Sigma.\ s \in P(n) \longrightarrow$ $(\lceil\rceil, s) \in \bigcap_{i=1}^{n} pre(A(S^*(i,n)))$
   (b) *Local correctness*: $\forall i.\ i \in \{1,\ldots,n\} \longrightarrow\ \vdash A(S^*(i,n))$
   (c) *Interference freedom*: $\forall i,j.$ if $i,j \in \{1,\ldots,n\}$ and $i \neq j$ then, for every assertion $r_k(i,n)$ in $A(S^*(i,n))$, and for every atomic action $a(j,n)$ with precondition $pre(a(j,n))$ in $A(S^*(j,n))$, the formula
   $$\vdash \{r_k(i,n) \cap pre(a(j,n))\}\ a(j,n)\ \{r_k(i,n)\}$$
   holds.
   (d) $\forall s^* \in \Sigma^*.\ s^* \in \bigcap_{i=1}^{n} post(A(S^*(i,n))) \longrightarrow s \in Q(n)$.

Then, using the rule for parameterized parallel programs, we can conclude that $\vdash_F \{P(n)\}\ z := \lceil\rceil; \|_{i=1}^{n} A(S^*(i,n))\ \{Q(n)\}$ holds. By deleting the assignments to auxiliary variables with the *reduction rule*[4] and ommiting the intermediate annotations we obtain $\vdash_F \{P(n)\}\ \|_{i=1}^{n} S(i,n)\ \{Q(n)\}$.

Note that with the system $O$ we could prove statements of the form

$$\begin{array}{l}\forall n.\ \models \{P(n)\}\|_{i=1}^{n} S(i,n)\{Q(n)\} \longrightarrow \\ \exists A^n.\ \vdash_O \{P(n)\}\|_{i=1}^{n} A^n(S^*(i,n))\{Q(n)\}\end{array}$$

where for different values $m$ and $l$ of $n$, $A^m$ could be very different from $A^l$. In this paper, however, we prove the existence of a uniform annotation (parameterized by $n$ and $i$) for all values of $n$, that is, we prove

$$\begin{array}{l}\models_F \{P(n)\}\|_{i=1}^{n} S(i,n)\{Q(n)\} \longrightarrow \\ \exists A.\ \vdash_F \{P(n)\}\|_{i=1}^{n} A(S^*(i,n))\{Q(n)\}\end{array}$$

### 3.1 Extending the Program

Let $V_t = \{x_1,\ldots,x_k,y_1,\ldots,y_l\}$ be the set of variables occurring in $S(i,n)$. Let $\overline{x}$ denote the coding $\lceil x_1,\ldots,x_k,y_1,\ldots,y_l\rceil$. Let $z$ be a new variable. Then, we transform $S(i,n)$ into another program $S^*(i,n)$ by replacing

**(i)** every **await**-statement **await** $b$ **then** $R$ **end**, where $R$ is not *skip*, by **await** $b$ **then** $z := z \frown \lceil i, \overline{x}\rceil; R$ **end**,

**(ii)** every assignment $y := t$ not inside an **await**-statement by **await** *true* **then** $z := z \frown \lceil i, \overline{x}\rceil; y := t$ **end**.

If our language allowed assignments to variables at every transition (including those consisting of evaluation of boolean conditions), completeness could be proven by extending the program with assignments to counter

[4]Standard rule of the Owicki-Gries system for the elimination of the auxiliary variables from the program code.

variables. In a parameterized program the counter variable would be a composed variable $c$. The $i$th component program would update this variable by updating its $i$th component, i.e. $c[i]$, at every transition. As we will see in §5, parallel programs are in practice verified using auxiliary variables as counters because finding assertions in terms of history variables is too difficult.

### 3.2 Annotating the Program

Given $S^*(i,n)$, our goal is to construct intermediate assertions[5] that yield a proof outline such that conditions (a), (b), (c) and (d) above can be proven.

By the completeness of $O$ we know that given

$$\models \{P(n)\}\|_{i=1}^{n} S(i,n)\{Q(n)\}$$

we can find for every value $m$ of $n$, i.e., for the particular valid triple

$$\models \{P(m)\}\ S(1,m)\|\ldots\|S(m,m)\ \{Q(m)\},$$

an extended program

$$z := \lceil\rceil; (S^*(1,m)\|\ldots\|S^*(m,m))$$

and an annotation $A^m$ so that

$$\vdash \{P(m)\}z := \lceil\rceil;\ \|_{i=1}^{m} A^m(S^*(i,m))\{Q(m)\}$$

$A^m$ associates to every point of interference $j$ inside a component $i$ an assertion (set of states) $r_{ji}^m$ whose construction is described in the completeness proof for the system $O$ [6, 7, 1]. Informally, this assertion is defined as the sets of states that are reachable by some computation starting in a state in the precondition, where the steps of the computation are determined by the rules of the semantics.

Now, we define the assertion associated with a location $j$ in a parameterized component program $S^*(i,n)$, and we call it $r_j(i,n)$, simply as follows:

$$r_j(i,n) = r_{ji}^n$$

where for each value $m$ of $n$ and for each value $l$ of $i$, the set $r_{jl}^m$ is the one constructed in the completeness proof of the Owicki-Gries system for the concrete parallel program $\|_{i=1}^{m} S(i,m)$ at location $j$ inside component $l$, $l \in \{1,\ldots,m\}$.

It remains to prove that with this annotation the conditions necessary to derive the corresponding triple using the system $F$ are satisfied. For this purpose, it suffices to prove that the conditions (a),(b),(c) and (d) mentioned above hold of this annotation. This is easy, since for any arbitrary value $m$ of $n$, the proofs of (a),(b),(c) and (d), reduce to the proofs for the concrete program with $m$ parallel components. We just show the proof of (a), the others are similar:

**(a)** Denote the first location (the one corresponding to the precondition) of a component by 0. Assume that there is a state $s$ such that $s \in P(m)$. By the completeness of the system $O$ we have $(\lceil\rceil, s) \in \bigcap_{i=1}^{m} r_{0i}^m$. By definition of $r_0(i,m)$, we have $(\lceil\rceil, s) \in \bigcap_{i=1}^{m} r_0(i,m)$.

This closes our proof of semantic completeness of the system $F$. By defining the intermediate assertions as parameterized functions that return for each $n$ and each $i$ the set of states that characterize the assertions of the particular program with those values of $n$ and $i$, we have proven that these sets of states exist for every $n$ and every $i$. The main goal of this paper, however, is to prove that these sets not only exist but can also be concretely described using an assertion language that includes first order arithmetic over the natural numbers.

## 4 Relative Completeness

In this section we prove that the semantic sets described in §3 can be syntactically expressed if we use an assertion language which contains at least the logical system of *elementary arithmetic* and consider programs whose boolean conditions and state transformations can be expressed in the assertion language. For this purpose we will use an important result of recursion theory [10]:

> For any relation $R$, if $R$ defines a recursively enumerable set, then $R$ is definable in elementary arithmetic.

Thus, it suffices to prove that the sets of states $r_j(i,n)$ that we constructed in §3 are recursively enumerable sets. Intuitively, a set $S$ is recursively enumerable (or checkable) if there exists a procedure that, given an element $t$ as input, "checks" whether $t$ is in $S$ and stops when its checking procedure succeeds. If $t$ is not in $S$ the procedure will continue checking forever. We show that such a procedure for checking the sets $r_j(i,n)$ exists.

Let $\|_{i=1}^{n} S(i,n)$ be a parameterized parallel program and let $P(n)$ be a set of initial states. For every value $m$ of $n$ and for every control point $j$ in component

[5] By an assertion we mean here a set of states as a semantic object.

$i$, the set $r_{ji}^m$ is recursively enumerable[6]. We denote the checking procedure by $M_{ji}^m$. Informally it works as follows:

A state $s$ is in $r_{ji}^m$ iff it is possible to start the execution of $S^*(1,m)\|\ldots\|S^*(m,m)$ with a state in $P(m)$ and reach the control point $j$ of component $i$ with variables as given by state $s$.

We now define the procedure $M_j$ that checks if a state $s$ is in $r_j(i,n)$:

It takes as arguments $n$, $i$ and the state $s$. With the values of $n$ and $i$, it can build the concrete program $S^*(1,n)\|\ldots\|S^*(n,n)$ and simulate the procedure $M_{ji}^n$. Then, $M_j(n,i,s)$ is defined such that it succeeds iff $M_{ji}^n(s)$ succeeds.

We can conclude that the domain of the procedure $M_j$ (the values for which it succeeds) is a recursively enumerable set. By the theorem mentioned above, the relation that defines this set is definable in elementary arithmetic. Let us recall what this means [10]:

> An $n$-ary relation $R$ is *definable in elementary arithmetic* if there is a formula $F\ a_1 \cdots a_n$ with free variables $a_1, \ldots, a_n$ such that
>
> $$R = \{(x_1, \ldots, x_n) \mid F x_1 \cdots x_n \textit{ is true}\}$$
>
> where, for any integers $x_1, \ldots, x_n$, $F\ x_1 \cdots x_n$ is the result of substituting the numeral expressions of $x_1, \ldots, x_n$ for $a_1, \ldots, a_n$, respectively, in $F\ a_1 \cdots a_n$.

The tuples of our set are formed by the values of $n$, $i$ and those of the program variables that form the state. Consequently, the formula defining this set will have as free variables not only the program variables (like for concrete programs) but also $n$ and $i$, as intuitively expected.

In the ticket algorithm, for example, the arithmetical expression defining some assertion would have as free variables, $n$, $i$, *num, next* and *turn*, but not references to the components of *turn*. In practice, however, to reason in Hoare style, more expressive logics than elementary arithmetic are used. If these logics allow to express composed variables as functions, expressions like $turn[i]$ can be written directly.

This completeness proof does not provide an effective methodology for finding the expressions for assertions. The important result is that these expressions always exist.

[6] Later, Apt proved that they were recursive sets [1] when the auxiliary variables were used as history variables, and only recursively enumerable if auxiliary variables were used as program counters.

# 5 Examples

There are several examples of verification of parameterized parallel programs in the style of Owicki and Gries in the literature [3, 2]. One of them is the ticket algorithm from Figure 1. The proof outline as given in [3],

$\{num = 1 \wedge next = 1 \wedge \forall i.1 \leq i \leq n \longrightarrow turn[i] = 0\}$
**while** *true* **do**
  $\{I\}$ $NCS(i,n)$;
  $\{I\}$ $\langle\ turn[i] := num;\ num := num + 1\ \rangle$
  $\{I\}$ **wait** $turn[i] = next$ **end**
  $\{I \wedge turn[i] = next\}$ $CS(i,n)$;
  $\{I \wedge turn[i] = next\}$ $next := next + 1$
**od**

where

$$I \stackrel{\text{def}}{=} next \geq 1\ \wedge\ \forall k,l.\ 1 \leq k \neq l \leq n\ \longrightarrow\ turn[k] < num \wedge\ (turn[k] = 0\ \vee\ turn[k] \neq turn[l]),$$

allows the verification of the program for any value of $n$ in the system $F$. The property of mutual exclusion holds because the conjunction of two or more assertions with different values for $i$ before entering the critical section implies *false*. These assertions represent the possible states at that point, consequently the set of states that can enter the critical section of two different components at the same time is empty, since no state satisfies *false*.

For the verification of the program $\{x = 0\} \|_{i=1}^n x := x + 1\ \{x = n\}$, auxiliary variables are unavoidable. We introduce an array variable $c$, such that the component $i$ of the parallel program atomically updates the shared variable $x$ and the $i$th component of $c$. All components of this array are first initialized to 0 (abbreviated by $c := 0$):

$$\begin{array}{l} \{x = 0\}\ c := 0 \\ \{x = 0\ \wedge\ \forall j.\ 1 \leq j \leq n \longrightarrow\ c[j] = 0\} \\ \|_{i=1}^n A(S^*(i,n)) \\ \{x = n\} \end{array}$$

A proof outline $A$ for the extended program $S^*(i,n)$ that satisfies the requirements for the verification in the system $F$ is

$$\begin{array}{c} \{c[i] = 0\ \wedge\ x = \sum_{j=1}^n c[j]\} \\ \langle x := x + 1; c[i] := 1\rangle \\ \{c[i] = 1\ \wedge\ x = \sum_{j=1}^n c[j]\} \end{array}$$

Again, we have not expressed the assertions as expressions in elementary arithmetic with free variables $n, i, x$ and $c$. However, projection and finite sumation are both partial recursive functions and as such can be expressed in a first order extension of arithmetic.

Both examples presented here have been verified in the theorem prover Isabelle [5] using the formalization of the system $F$, where the logic used for expressing the assertions is HOL (Higher Order Logic). This logic allows a very high level of expressivity. A larger example, namely, a garbage collection algorithm for a parameterized number of mutators, has also been verified in Isabelle [9].

## 6 Conclusions

Recently the verification of parameterized systems has been a main topic of research. We have aimed at exploring the limits of assertional reasoning, where the user is requested to characterize the system behavior by logical assertions and establish validity by a formal proof. Our original motivation was to study if it is always possible to find such a proof for any valid specification of a parameterized program.

The Owicki-Gries system is the most fundamental assertional method for reasoning about the correctness of parallel programs. However, it was not designed for the verification of parameterized parallel programs, but only for programs with a given number of components. We have shown that the system can be generalized to deal with parameterized parallel programs and that the resulting system is sound and (relatively) complete.

From the completeness of the original Owicki-Gries system we know that for each instance of a valid specification of a parameterized parallel program there exists a proof in the system. The main result presented in this paper, namely, the completeness of the extended system, states that it is always possible to find a single proof that works for any number of components. Not surprisingly (but not obviously) the assertions that allow to derive this proof in the system are themselves parameterized. Then, the particular proof for any instance of the parameterized program can be obtained by instantiating the parameters in the assertions.

The main difficulty of this work lies in a satisfactory formalization of the problem. We have given rigorous definitions of the syntax and semantics of parameterized parallel programs that make them suitable for their study in a Hoare logic framework. Thus, we believe that the results presented in this paper can be easily adapted to other Hoare-logic verification systems like, for example, the Rely-Guarantee method.

## References


[1] K. R. Apt. Recursive assertions and parallel programs. *Acta Informatica*, 15:219–232, 1981.

[2] E. Best. *Semantics of Sequential and Parallel Programs.* International Series in Computer Science. Prentice Hall, 1996.

[3] W.-P. de Roever, F. de Boer, et al. *State-Based Proof Theory of Concurrency: from Noncompositional to Compositional Methods.* Cambridge University Press Tracts in Theoretical Computer Science, 2000. To appear.

[4] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[5] T. Nipkow and L. Prensa-Nieto. Owicki/Gries in Isabelle/HOL. In J.-P. Finance, editor, *Fundamental Approaches to Software Engineering (FASE'99)*, volume 1577 of *LNCS*, pages 188–203. Springer-Verlag, 1999.

[6] S. Owicki. *Axiomatic proof techniques for parallel programs.* PhD thesis, Computer Science Dept., Cornell University, 1975.

[7] S. Owicki. A consistent and complete deductive system for the verification of parallel programs. In *Proceedings of the 8th Annual Symposium on Theory of Computing*, pages 73–86, 1976.

[8] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[9] L. Prensa-Nieto and J. Esparza. Verifying single and multi-mutator garbage collectors with Owicki/Gries in Isabelle/HOL. In M. Nielsen and B. Rovan, editors, *Mathematical Foundations of Computer Science (MFCS 2000)*, volume 1893, pages 619–628, 2000.

[10] H. Rogers, Jr. *Theory of Recursive Functions and Effective Computability.* MIT Press, 1987.

[11] J. R. Schoenfield. *Mathematical Logic.* Addison-Wesley, 1967.



**Acknowledgment** I would like to thank Javier Esparza, Tobias Nipkow and David von Oheimb for helpful and insightful discussions.