Automating the Generation of Program Analysis and Verification Tools

Leonardo Mendonça de Moura
e-mail: moura@les.inf.puc-rio.br

Carlos José Pereira de Lucena
e-mail: lucena@inf.puc-rio.br

Edward Hermann Haeusler
e-mail: hermann@inf.puc-rio.br

PUC-RioInf.MCC08/00 February, 2000

Abstract: The use of domain-specific languages (DSLs) is an interesting alternative for software development. DSLs increases productivity, reliability, and flexibility. However, implementing them is often difficult and costly. The problem is even worse if simulation, analysis and verification tools are desired. Presently, program analysis and verification is at best achieved by a hand-crafted tool specific to a programming/specification language. Since the manual construction of tools is expensive, they are hard to obtain and not often used, limiting the quality of the implemented software. We have proposed a framework for the automated generation of simulation, analysis and verification tools for programs/specifications based on machine-processable formal definitions of the language's operational semantics. We also use a modular approach for language semantics that allows us to reuse semantic modules and fragments of already defined tools. In this article, the theoretical foundations and algorithms of our framework are described. Although, the main motivation for our framework is to automate the development of program analysis and verification tools for DSLs, it can also be used in the development of such tools for general-purpose languages.

Keywords: Program Analysis and Verification, SOS, DSLs.

Resumo: Linguagens específicas de domínio (DSLs) constituem um alternativa interessante para o desenvolvimento de software. DSLs aumentam a produtividade, robustez e flexibilidade no desenvolvimento de software. Entretanto, a implementação destas linguagens é em geral difícil e custosa. O problema é ainda mais grave se ferramentas de simulação, análise, e verificação são necessárias. Atualmente, análise e verificação de programas é realizada por ferramentas específicas construídas manualmente. Já que a construção destas ferramentas é cara, estas são difíceis de serem obtidas e raramente utilizadas. Portanto, neste trabalho, nós propomos um framework para a geração automática de ferramentas de simulação, análise, e verificação. Nós utilizamos uma abordagem modular, que permite a reutilização de módulos semânticos. O nosso framework pode também ser utilizado para a construção de ferramentas para linguagens de propósito geral.

Palavras-chave: Análise e Verificação de Programas, SOS, DSLs.

1 Introduction

Recently there has been increased interest in the so-called *domain-specific languages* (DSLs) [Bel94, Hud96, ABBC99, CRL99], *i.e.* languages focused on specific application domains. A successful DSL is one which captures the native vocabulary and semantics of the application domain. Programs coded in a DSL can be considered specifications, if we understand specifications as higher-level descriptions. The use of DSLs consists in an interesting alternative for software development. The advantage of using DSLs is that the domain expert can express domain specific concepts directly rather than encoding them in a general purpose language. The user's intentions remain clear and are not blended in a low level language. Because constructs in a DSL abstract key concepts of the domain, the developer can write more concise and higher level programs in less time. Another major motivation for domain-specific languages is their ability to support end-user programming. DSLs often employ familiar notation that permits subject-matter experts to sidestep professional programmers. DSLs not only capture domain abstractions in a reusable fashion, they can also support analysis and verifications that would be infeasible in the context of *general-purpose languages* (GPLs). More sophisticated analysis can be used to support new optimizations and transformations improving efficiency and quality. Software development models based on DSLs are primarily aimed at achieving faster development of safer applications.

Though actual uses of DSLs brings about benefits such as productivity, reliability, and flexibility, implementing DSLs is often difficult and costly [Cle88]. It is wise to note that the implementation of a new programming language is not limited to construction of a interpreter or compiler. Tools for testing, verifying, analyzing and visualizing code are also desired. Presently, program analysis and verification is at best done by a hand-crafted tool specific to a programming/specification language. Since the manual construction of tools is expensive, they are hard to obtain and not often used, limiting the quality of implemented software.

We have proposed a framework for the automated generation of simulation, analysis and verification tools for programs/specifications based on machine-processable formal definitions of language semantics.

The framework relies on state-exploration techniques based on theories of *Abstract Interpretation* [CC77, Cou96] and *Model Checking* [CES86, Hol91]. The relationship between flow analysis (abstract interpretations) and model checking was first proposed by Steffen [Ste91]. This relationship was also explored by Schmidt [Sch98].

The main motivation for our work was the Design Maintenance System [Bax92] (DMS). DMS is fundamentally a transformational synthesis system, i.e. a semi-interactive system for deriving code from specifications by repeatedly applying transformations. The transformation knowledge is organized around knowledge clusters called *domains*. This domain network provides all the knowledge needed for mechanically constructing the software. It provides application *domain specific languages* (DSLs) to describe the software specifications, software components, transformations to derive lower level descriptions from higher-level descriptions, analyzers and verifiers to validate and simulate descriptions at different levels of abstraction.

The work described here adds strong analysis and verification capabilities to DMS enabling the software engineer to understand, develop and modify a software system by providing relevant information to support such activities. Additionally, the static analysis information produced by our system helps DMS in the derivation process. It is important to notice that our framework may be used independently of DMS, or associated with other transformational systems such as Draco [LSF94] or TXL [CC93].

In this article, the theoretical foundations and algorithms of our framework are described. We also describe an application of our framework in the domain of industrial process control and programmable logic controllers (PLCs). The choice of the example was motivated by the work described in [AT98]. In that work the author proposed that the design of DSLs should include the development of a proof system.

Although, the main motivation of our framework is to automate the development of program analysis and verification tools for DSLs, it can also be used in the development of such tools for general-purpose languages.

2 The Framework

Our framework automates the generation of simulation, analysis and verification tools starting from the operational semantics of a DSL or GPL. The operational semantics of a given programming language defines the runtime behavior of programs, and it shall also be called "concrete" semantics or "concrete" interpretation. One way to think about the operational semantics of a language is to regard it as defining an interpreter for such language. For now, we will assume that the operational semantics of a language defines a function $\mathit{successors} : \mathit{State} \rightarrow \wp(\mathit{State})$, *i.e.* a function that given a state returns all possible successors of such state. We define *trace* as a finite or infinite sequence of states $(s_0, s_1, \ldots, s_i, \ldots)$, where $s_{i+1} \in \mathit{successors}(s_i)$, *i.e.* there is a transition from s_i to s_{i+1}. Briefly, an interpreter defines all possible *traces* of a given program. The set of *traces* of a particular program is called a *state space* of the program. We shall also use $s_i \rightarrow s_j$ meaning $s_j \in \mathit{successors}(s_i)$. The reflexive and transitive closure of the relation $\rightarrow$ is written as $\twoheadrightarrow$.

For deterministic sequential languages such as *Pascal*, given a program and a fixed input data, the state space is simply a sequence of states. If a program has an infinite number of possible inputs, then the state space is represented by a possibly infinite set of sequence of states, *i.e.* a sequence of states for each input data. For nondeterministic languages, the state space of a given program is represented by a tree, where each node is a state, and node's children are the possible successor states. Surely such *state spaces* are usually infinite structures which may not be explicitly represented in a computer.

In our framework, all program simulation, analysis and verification tools are based on algorithms that traverse the state space of programs. The nature of such algorithms will determine the kind of tool we are generating. In a few words, each kind of tool "interprets" a program in a different way. Simulators simply interpret a program, possibly using some kind of debugging support. Analyzers "interpret" a program to "collect" runtime information that can be used to infer invariants or properties. Verifiers "interpret" a program to "check" assertions or temporal claims violations.

```
simulate (State s0)
{
  State s = s0;
  while (successors(s) ≠ ∅)
        s = select(successors(s));
}
```

Figure 1: The simulation algorithm

2.1 Simulators

Simulators are the simplest kind of tool that might be generated in our framework. They simply execute a particular *trace* of the program. The framework provides infrastructure to define different "flavors" of simulation starting from the operational semantics. The framework supports the following "flavors" of simulation:

Random Selection: the successor state is chosen randomly. For deterministic languages this "flavor" behaves as a standard interpreter.

Guided: the user chooses the successor state.

Oracle: the user provides a function that chooses the successor state. Notice that such function may request user interaction in specific points of the simulation.

Abstractly, all "flavors" described above can be seen as instances of the algorithm described in Figure 1. The only difference is the way the function *select* is implemented.

If desired, the user can also select simulation algorithms that store the selected trace during the simulation, saving the *history* of the computation. We also provide support for common debugging operations like setting breakpoints and inspecting the value of program variables. Obviously, none of such features are described in Figure 1.

2.2 Analyzers

The primary objective of analysis algorithms is to collect information about runtime behavior of a program. Compilers usually use such kind of information to apply optimizations such as loop-invariant code motion, common-subexpression elimination and dead code elimination [Muc97]. The produced information can also help the user to construct, understand and debug programs. For instance, a program analyzer can be applied in techniques such as program slicing [Wei84].

State spaces of programs are usually infinite structures which can not be explicitly represented in computers. Therefore, approximations of language semantics must be used, in which concrete or exact properties are replaced by abstract or approximate properties, e.g. data like 2 and -3 are replaced by *pos* (positive) and *neg* (negative), respectively. The process of "executing" a

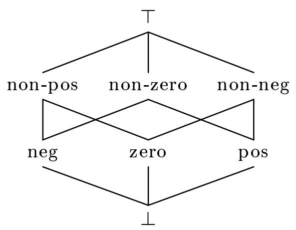

Figure 2: Lattice *Sign*

program using abstract data is called *abstract interpretation*. It is convenient for us to see abstract interpretation as a kind of "symbolic execution".

An abstract interpretation of a program produces an *abstract state space*. Nondeterminism is always present in abstract interpretations where loss of precision makes many language operators nondeterministic. For instance, the statement

$$\text{if (x > 10) x = x + 1}$$

can not be deterministically executed, if at runtime the value of variable x is *pos*. Nevertheless, if the value is *neg* the body of statement *if* will certainly not be executed.

We illustrate these concepts with the *Sign* analyzer. In sign analysis, one is interested in estimating the sign of the value of an expression in the absence of the actual values assumed by variables in the expression. The domain of abstract values is described by the lattice in Figure 2. For instance, *non-neg* represents the set $\{0, 1, 2, \ldots\}$, *neg* the set $\{-1, -2, \ldots\}$, and $\top$ the set $\{\ldots, -1, 0, 1, \ldots\}$. In such lattice, element $\top$ represents the property "anything" (or the answer "I do not know"), and it is always used when we are not able to infer the sign of a variable due to imprecisions in the approximated semantics.

Let us consider the program *Factorial* described in Figure 3. The program contains several labels (*L1, L2, ...*) which the only purpose is to simplify our description of the abstract state space. Figure 4 contains the abstract state space of such program. This example illustrates the use of graphs to represent *regular* infinite state spaces. A state space is regular if every path contains duplicated nodes. Notice that such graph is similar to a control-flow graph [Muc97] decorated with abstract properties.

All abstract values sets have a partial order relation ($\sqsubseteq$). The relation $\sqsubseteq$ captures the notion of being a better approximation. Considering $x \sqsubseteq y$, we say that x is a better approximation than y. For every x we have $x \sqsubseteq \top$, *i.e. top* is the worst approximation. For our purposes, every abstract values set should be at least a partial order with a top ($\top$) element which represents the property "anything". Additional structure is necessary for some analyzers. For example, some analysis algorithms require the abstract values set to be a lattice, *i.e.* a set with operations $\sqcup$ (*join*) and $\sqcap$ (*meet*), which satisfy several properties, as follows:

1. $\forall x, y \in L\ \exists z, w \in L \cdot x \sqcup y = z \textit{ and } x \sqcap y = w$

2. $\forall x, y \in L \cdot x \sqcup y = y \sqcup x \textit{ and } x \sqcap y = y \sqcap x$

3. $\forall x, y, z \in L \cdot x \sqcup (y \sqcup z) = (x \sqcup y) \sqcup z \textit{ and } x \sqcap (y \sqcap z) = (x \sqcap y) \sqcap z$

```
INPUT x;
y = 1;
L1:
while (x > 1)
{ L2: y = y * x;
  L3: x = x - 1;
  L4:
}
L5:
```

Figure 3: Program Factorial

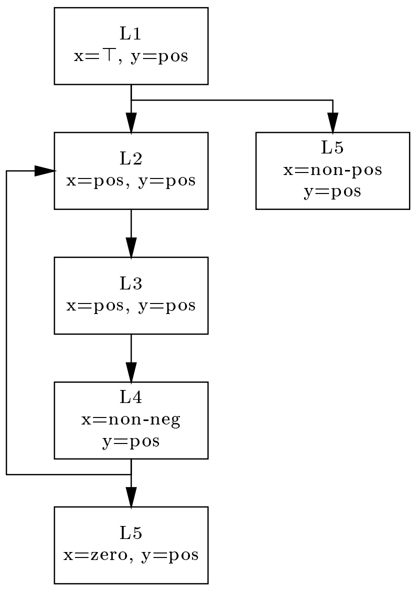

Figure 4: *Factorial* Abstract State Space

4. The elements $\perp$ (*bottom*) and $\top$ (*top*) exist such that $\forall x \in L \cdot x \sqcap \perp = \perp \ and\ x \sqcup \top = \top$

Of course every lattice is a partial order, and the relation $\sqsubseteq$ is defined as $x \sqsubseteq y \Leftrightarrow x \sqcap y = x$. The operator $\sqcup$ is often used to combine "information".

The concept of *safe simulation* [Sch98] is formalized as a homomorphism from the "*concrete*" state space into the abstract state space. The intuitive rationale for this concept is that any transition in the "concrete" state space is mimicked by one in the abstract state space, *i.e.* any possible behavior of the program is simulated by the abstract interpretation. However, the abstract state space might also contains traces that are not present in the program's concrete state space.

Define a function $\alpha_{state} : State \rightarrow AbsState$ from states to abstract states. For instance, sign analysis α_{state} maps integer values associated with variables into elements of lattice *Sign* such as *pos* and *neg*. Now, the homomorphism from the concrete state space into the abstract state space is defined more formally as:

$$\forall s, s' \cdot s \rightarrow s' \Rightarrow \exists a' \cdot \alpha_{state}(s) \rightarrow a' \ and\ \alpha_{state}(s') \sqsubseteq a'$$

Informally for each concrete transition $s \rightarrow s'$ exists an abstract state a', and an abstract transition from the abstraction of s to a' ($\alpha_{state}(s) \rightarrow a'$), such that a' is an approximation of s' ($\alpha_{state}(s') \sqsubseteq a'$). Notice that we do not require a' to be the best approximation of s' ($\alpha_{state}(s') = a'$).

We shall say that an abstract state *sa* is a "safe" approximation of a concrete state *sc* if:

$$sc\ safe_{state}\ sa \Leftrightarrow \alpha_{state}(sc) \sqsubseteq sa$$

As mentioned before, the state space of a program may be represented by a tree. For a state space (tree) t, we shall write $root(t)$ to denote the root of t. We write $t \rightarrow t'$ as a syntactic sugar for $root(t) \rightarrow root(t')$. Using these definitions, we say that a concrete state space is safely approximated by an abstract state space if:

$$\begin{array}{rcl} t\ safe_{tree}\ t' & \Leftrightarrow & root(t)\ safe_{state} root(t'),\ \text{and for all transition}\ t \rightarrow t_i, \\ & & \text{exists a transition}\ t' \rightarrow t'_j, \text{such that}\ t_i\ safe_{tree}\ t'_j \end{array}$$

Informally this relation states that every path in the concrete state space is safely simulated (mimicked) by a path in the abstract state space. It is wise to note that this is a co-recursive definition, and we must use co-inductive arguments, once state spaces might be infinite structures, and we are interested in the greatest fix point [1]. It is not surprising the use of co-inductive arguments, if we remember that the definition of $safe_{tree}$ is extremely similar to the definition of *Bissimilarity* [Mil89] used in observational equivalence. Thus, we may say that $safe_{tree}$ is an "approximated" equivalence relation.

Using these definitions, we can say that an abstract state space is a safe approximation if the following property holds:

$$\forall s, s' \forall a \cdot s \rightarrow s' \ and\ s\ safe_{state}\ a \Rightarrow \exists a' \cdot a \rightarrow a' \ and\ s'\ safe_{state}\ a'$$

[1] The least fix point is the empty relation, that is useless for our purposes.

Such property can be easily demonstrated as equivalent to the homomorphism definition given above, if $safe_{state}$ is *U-closed* and the abstract transition relation is monotonic. Notice that $safe_{state}$ is *U-closed* if $s\ safe_{state}\ a\ and\ a \sqsubseteq a' \Rightarrow s\ safe_{state}\ a'$. An abstract transition relation is monotonic if:

$$\forall a_1, a_1', a_2 \cdot a_1 \rightarrow a_1' \ and\ a_1 \sqsubseteq a_2 \Rightarrow \exists a_2' \cdot a_2 \rightarrow a_2' \ and\ a_1' \sqsubseteq a_2'$$

It is usually trivial to prove that $safe_{state}$ is *U-closed* and the abstract transition relation is monotonic. So the safety of a simulation is demonstrated by showing that for every concrete state s, an approximation a of s, and for every transition $s \rightarrow s'$ exists a transition $a \rightarrow a'$ such that a' is an approximation of s'.

In the "classical" approach for abstract interpretation [CC77] the safety is ensured by the fact that *post fix points* [2] are approximation of least fix points. It is also used the concept of *galois connection* defined as:

Definition 1 *If* $(L_1, \sqsubseteq_1)$ *e* $(L_2, \sqsubseteq_2)$ *are lattices, then* (α, γ) *is a* Galois Connection *between* L_1 *and* L_2*, if and only if,* $\alpha \in L_1 \rightarrow L_2$ *and* $\gamma \in L_2 \rightarrow L_1$ *are functions such that:*

$$\forall x \in L_1, y \in L_2 \cdot \alpha(x) \sqsubseteq_2 y \Leftrightarrow x \sqsubseteq_1 \gamma(y)$$

The "classical" approach may be related to our approach, if we note that if the abstract values set is a complete lattice and the property $c\ safe_{state} \sqcap \{a' \mid c\ safe_{state}\ a'\}$ holds, then $safe_{state}$ defines a *galois connection* between $\wp(State)$ and $AbsState$ ($\alpha : \wp(State) \rightarrow AbsState$ and $\gamma : AbsState \rightarrow \wp(State)$).

$$\begin{aligned} \alpha(S) &= \bigsqcup_{c \in S} \{\sqcap\{a \mid c\ safe_{state}\ a\}\} \\ \gamma(a) &= \{c \mid c\ safe_{state}\ a\} \end{aligned}$$

A crucial issue in program analysis is termination. If we desire to infer assertions and properties of a program, the respective abstract state space must be a safe approximation and a finite data structure. Therefore, we prove termination by showing that the abstract state space may be finitely represented by a data structure.

Our framework provides several algorithms to perform program analysis. All of them are based on the abstraction of function *successors* (*i.e.* semantic approximations). The user must also provide functions to compare states, to copy states, and to produce hashcodes. Some algorithms also require the definition of operators such as $\sqcup$. Of course, the definition of the abstraction of function *successors* is derived from the concrete one.

2.2.1 A Simple Algorithm

If the abstract state space is a finite structure due to the use of semantic approximations, then we might use a two-step algorithm to compute the analysis information. For example, sign analysis for the simple imperative language described above always produces a finite abstract state space. In the first step, the finite abstract state space is computed by using the algorithm described in Figure 5. Such algorithm simply computes all possible abstract "traces", and

[2] a is a post fix point of F iff $F(a) \sqsubseteq a$

```
visited-states = ∅; worklist = {s0};
while (worklist ≠ ∅)
{
  curr-state = remove-an-element(worklist);
  visited-states = visited-states ∪ {curr-state};
  for each succ-state in successors(curr-state)
  {
    connect(curr-state, succ-state);
    if (succ-state ∉ visited-states)
    { worklist = worklist ∪ {succ-state}; }
  }
}
```

Figure 5: Simple Analysis Algorithm

stores them in a graph. Variable *visited-states* contains the set of already visited states. Variable *worklist* contains the set of states not visited yet. Function *remove-an-element* removes and returns an element from a set. Function *connect* is used to connect nodes of the abstract state space. The implementation of *worklist* defines the kind of search which will be performed. In our framework, we implemented *worklist* using a stack discipline. So the algorithm performs a depth-first search.

Once the abstract state space is constructed, analysis information is extracted from it. The extracted information is called *collecting semantics.* The "classical" collecting semantics is *state-based.* For example, in sign analysis we are often interested in determining the sign of a particular variable in a specific program-point. This kind of analysis information can be directly extracted from the abstract state space by using algorithms that traverse it. Abstractly, given an abstract state space t and a program-point pp, the analysis information is computed using a function such as:

$$\mathit{ana\text{-}info} : \mathit{Abstract\text{-}State\text{-}Space} \times \mathit{Program\text{-}Point} \rightarrow \wp(\mathit{Abstract\text{-}State})$$

$$\mathit{ana\text{-}info}(t, pp) = \{n \mid n \in t \ \mathit{and} \ \mathit{prog\text{-}point}(n) = pp\}$$

where function $\mathit{prog\text{-}point} : \mathit{Abstract\text{-}State} \rightarrow \mathit{Program\text{-}Point}$ returns the program-point associated with a state. The operator $\sqcup$ can be used to produce a single abstract state space that approximates the set $\wp(\mathit{Abstract\text{-}State})$. This way, we might redefine function *ana-info* as:

$$\mathit{ana\text{-}info}(t, pp) = \bigsqcup\{n \mid n \in t \ \mathit{and} \ \mathit{prog\text{-}point}(n) = pp\}$$

In the abstract state space t_{fact} of Figure 4, the program-point *L5* is associated with two states. Hence it becomes:

$$\begin{aligned} \mathit{ana\text{-}info}(t_{fact}, L5) &= \bigsqcup\{\langle x = \mathit{non\text{-}pos}, y = \mathit{pos}\rangle, \langle x = \mathit{zero}, y = \mathit{pos}\rangle\} \\ &= \{\langle x = \mathit{non\text{-}pos}, y = \mathit{pos}\rangle\} \end{aligned}$$

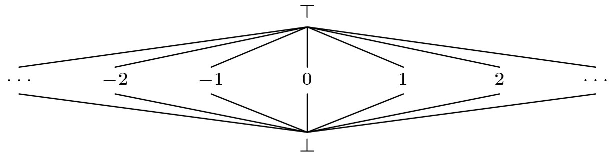

Figure 6: Lattice *Constant*

Several analysis such as reaching-definitions and live-variables [Muc97] are not state-based, because they involve path information. Briefly, the collecting algorithms must compute information about paths in the abstract state space. Abstractly, such algorithm is described as:

$$\textit{ana-info} : \textit{Abstract-State-Space} \times \textit{Program-Point} \rightarrow \wp(\textit{Abstract-Paths})$$

$$\textit{ana-info}(t, pp) = \{p \mid p \textit{ is a path from } s_0 \textit{ to } s_i \textit{ and prog-point}(s_i) = pp\}$$

The above algorithm reflects the so called forward data flow analysis algorithms such as reaching-definitions [Muc97]. Backward data flow analysis algorithms such as live-variables are abstractly described as:

$$\textit{ana-info}(t, pp) = \{p \mid p \textit{ is a maximal path from } s \textit{ and prog-point}(s) = pp\}$$

All path-based algorithms are extremely similar to the ones used in *model checking* of μ-calculus formulas. The model checking algorithms are described in [Arn92], and such relationship was first proposed by Steffen [Ste91] and was also explored by Schmidt [Sch98].

In our framework, we also provide algorithms that computes the two-step process in a single one. For all state-based analyzers, the abstract state space is constructed in such way that all states associated with the same program-point are merged, *i.e.* if the program-point associated with *curr-state* is equal to the program-point of an already visited state, then the states are merged using $\sqcup$ operators. However, as noted by Schmidt [Sch98] precision may be lost inadvertently when one formulates an analysis algorithm that merges both steps in a single one.

2.2.2 Algorithm for infinite abstract state spaces

Some analyzers use approximations that do not ensure finiteness of the abstract state space. For example, constant-propagation analysis uses a set of abstract values described by the lattice of Figure 6. Even, a simple program such as

```
x := 1;
while(true) x := x+1;
```

does not have a finite abstract state space [3]. Therefore, we can not use the algorithm described in Figure 5 to perform constant-propagation analysis.

To ensure the termination of the analysis, we provide algorithms that apply additional approximations during the construction of abstract state spaces. Such approximations are orthogonal to the semantic approximations described

[3]Actually, this program has a huge finite abstract state space, if the values of integer variables are bounded as usual.

by the user, *i.e.* they are analysis independent. In constant-propagation analysis, we ensure termination, if states associated with the same program point are "merged", using $\sqcup$ (join) operators. Such technique ensures termination because the lattice of abstract values has finite height.

Some analysis (e.g. interval analysis) use lattices with infinite height. For instance, interval analyzer estimates the interval of values that an integer variable can assume in a given program point. Therefore, we ensure termination by using $\bigtriangledown$ (widening) operators [CC92]. We could view the widening operator as an "aggressive" version of the join operator. The widening operator sacrifices precision in order to ensure termination. We also use $\triangle$ (narrowing) operators [CC92] to refine the imprecisions produced by widening operators.

Figure 7 describes an algorithm that uses $\sqcup$ during the construction of abstract state spaces. Notice that $\sqcup$ may be substituted by $\bigtriangledown$. As mentioned before, the use of $\sqcup$ during the construction of abstract state spaces may imply in loss of precision. Therefore, our algorithm tries to reduce the number of times $\sqcup$ (or $\bigtriangledown$) is applied. Variable *worklist* is a list of pairs $\langle state, path\rangle$, where *path* contains a sequence of states in the current path (trace) since the last time $\sqcup$ was applied. Function *cons* add an element to a list. Function *search-prog-point(path, pp)* looks for a state associated with program-point *pp* in the list *path*, and it returns *null* if such state does not exist. Notice also that $\sqcup$ (or $\bigtriangledown$) operators are only applied when a given program point is visited again.

We illustrate the algorithm by describing an interval analyzer. As stated above, a widening operator must be used to ensure termination in interval analysis. The widening operator is defined as:

$$[l_1, u_1] \bigtriangledown [l_2, u_2] = [\text{if } l_2 < l_1 \text{ then } -\infty \text{ else } l_1, \text{if } u_2 > u_1 \text{ then } +\infty \text{ else } u_1]$$

Using this algorithm over the program described in Figure 8, we obtain the abstract state space described in Figure 9. The performance of the algorithm can be improved if we always apply $\bigtriangledown$, *i.e.* if the program-point associated with *curr-state* is equal to the program-point of an already visited state, then the states are merged using $\bigtriangledown$, and the result is inserted again in *worklist*. However, we shall also lose precision. Figure 10 contains the abstract state space computed by using the algorithm with improved performance. Clearly, the second abstract state space is less precise. For example, in the second abstract state space we can only infer that $x \geq 11$ at program point $L6$, while in the first one we can infer that $x \geq 12$.

2.2.3 Interprocedural Analysis

Interprocedural [4] analysis is frequently necessary to achieve reasonable results. Otherwise, the analysis would have to conservatively assume the worst-case scenario for a call to an unknown procedure (function, method, ...).

Recursion complicates interprocedural analysis in general, since multiple instantiations (invocations) of recursive procedures may be active simultaneously. Somehow the analysis must abstract such potentially infinite set of procedure instantiations. Again, the idea is to "join" (merge) related procedure instantiations to ensure termination. However, the use of "join" operations may imply in

[4] We believe that interabstraction analysis is a more appropriate name, where abstraction means functions, procedures, methods, etc. However, we keep the name interprocedural analysis, since it is frequently used in program analysis literature.

```
visited-states = empty-list;
worklist = cons(⟨s0, empty-list ⟩, empty-list);
while (worklist ≠ empty-list)
{
  ⟨ curr-state, curr-path ⟩ = remove-first(worklist);
  visited-states = cons(curr-state, visited-states);
  curr-path = cons(curr-state, curr-path);
  for each succ-state in successors(curr-state)
  {
    if ¬(is-in(succ-state,visited-states))
    {
      same-prog-point-state = search-prog-point(curr-path,
                                                prog-point(succ-state));
      if (same-prog-point-state = null)
      {
        connect(curr-state, succ-state);
        worklist = cons(⟨ succ-state, curr-path ⟩, worklist);
      }
      else
      {
        join-state = same-prog-point-state ⊔ succ-state;
        connect(curr-state, join-state);
        if ¬(is-in(join-state, visited-states))
        { worklist = cons(⟨ join-state, empty-list⟩, worklist); }
      }
    }
    else { connect(curr-state, succ-state); }
  }
}
```

Figure 7: Analysis Algorithm

```
INPUT x;
L1: while (x ≥ 10)
{
  L2: if (x == 10)
      { L3: y = 2; }
      else
      { L4: y = 1; }
  L5: x = x + y;
  L6:
}
L7:
```

Figure 8: Program P_1

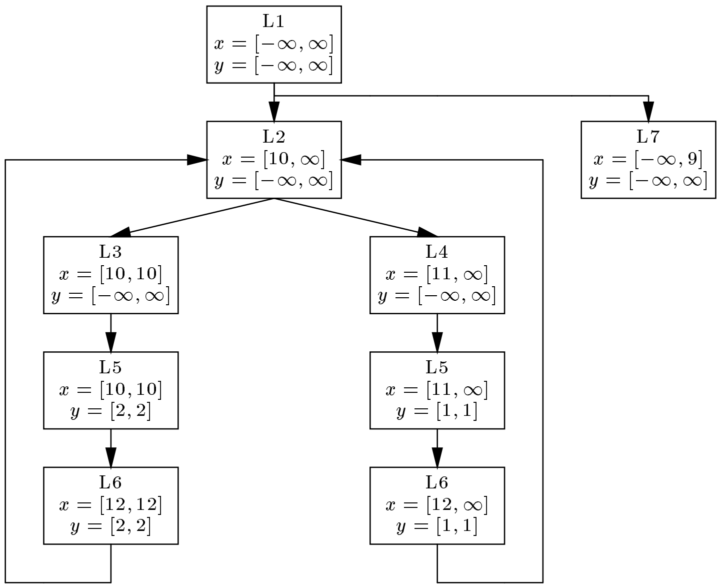

Figure 9: Program P_1 abstract state space

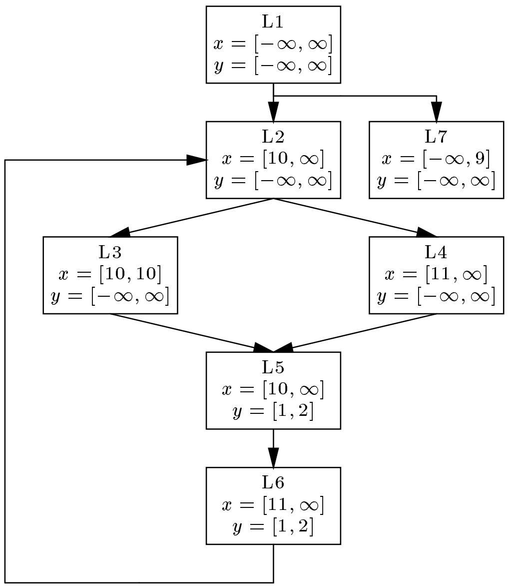

Figure 10: Program P_1 abstract state space

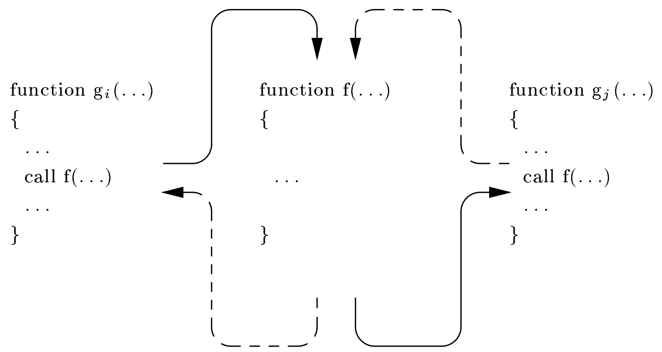

Figure 11: Unrealizable path

inadvertently loss of precision. For instance, suppose that we "join" all procedure invocations associated with a procedure f, if f is called from several other procedures g_i, then we will be considering several unrealizable paths as the one (dotted-line) described in Figure 11.

We minimize such problem by allowing the user to define a function *should-merge* that "decides" whether two procedure instantiations should be merged. Extremely simple functions *should-merge* produce reasonable results. For example, a function *should-merge* that "joins" two procedure instantiations, if the n most recent call sites are equal. Such function reflects the *nCFA* [Shi88] approach for interprocedural analysis. A more complex function *should-merge* may use information associated with the values of procedure arguments and global variables.

2.2.4 Using Analysis Tools

As stated above, the main use of analysis tools is to support the application of program optimizations and transformations. The information collected by an analysis tool allows us to verify whether a particular transformation may be applied. Abstractly, a transformation may be written as:

$$\textit{code} \Rightarrow \textit{code'}, \quad \textit{if condition } c \textit{ is satisfied}$$

A simple transformation is:

$$x/x \Rightarrow 1, \quad \textit{if } x \neq 0$$

In this example, the condition $x \neq 0$ must be satisfied, since $x = 0$ such transformation will modify the program semantics [5]. A sign analyzer allows us to verify whether condition $x \neq 0$ is valid at a given point of the program.

At first sight, we may think that nobody will ever write something "stupid" as x/x, and so such kind of optimizations would be useless. However, the use of encapsulation techniques (functions, procedures, ...) may hide the presence of "stupid" code. For instance, consider a function to solve equations like:

$$ax^2 + bx + c = 0$$

[5] We are assuming that a division by zero throws an exception

```
solve(float a, float b, float c)
{
  tmp1 := b*b - 4*a*c;
  if (tmp1 < 0)
  { throw exception "no solutions"; }
  tmp1 := sqrt(tmp1) / (2 * a);
  tmp2 := - b / (2 * a);
  return ⟨ tmp2 - tmp1, tmp2 + tmp1 ⟩;
}
```

Figure 12: Function *Solve*

It is well known that such equation has the following solutions:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Figure 12 contains a function that solves such kind of equation. This function receives the values of a, b and c, and returns both solutions, if they exist. Consider also a call site such as `solve(a,2*a,c)`, then if function *solve* is unfolded, we will have, for example, `tmp2 := -(2*a)/(2*a);`, that is an instance of the optimization described above. The point here is that the context in which a function is used may open new optimization possibilities. Moreover, for DSLs the context in which language primitives are used may also allow us to apply new optimizations.

Now, we describe several examples of optimization transformations based on information provided by the analysis tools.

Compilers usually generate code to check whether array indices are correct at runtime. Interval analysis may be used to prove that every array access in a program is statically correct, so compilers need not to generate the checking code. This kind of optimization is extremely common, and may be abstractly represented in the following way. Consider a language primitive or user-defined function P described in Figure 13. If a program analysis tool infers that condition $cond_i$ is always valid in a given point of the program, then a customized version of P may be automatically generated by removing the statement *if* associated with $cond_i$. Obviously, such optimization is more effective when applied to language primitives, once the structure of conditions $cond_i$ is known in advance, and a specific analyzer may be developed to check them. Therefore, this kind of optimization is particularly interested in the context of DSLs, where a much richer set of language primitives is available, and domain specific analysis tools may be developed.

When abstractions using functions or classes of general-purpose languages (GPLs) are defined, the compiler, in principle, has no knowledge about algebraic properties associated with them. Such algebraic properties are often used when we develop new program transformations. In GPLs, most of integer based optimizations relies on algebraic properties such as $n+0 = n$ or $n/n = 1$, $if\, n \neq 0$. In languages such as *Lisp*, we may also consider algebraic properties of *lists*, such as $car(cons(h,t)) = h$, to develop optimizing transformations. In other words, the

```
implementation of P
{
  if (¬cond_1)
  { throw exception ex_1 }
  ...
  if (¬cond_n)
  { throw exception ex_n }
  do something ...
}
```

Figure 13: Language primitive or user-defined function P

algebraic properties of the predefined (primitive) data-types of a programming language are often useful to develop program transformations. This observation is particularly interesting in the context of DSLs, in which richer domain specific data types are predefined.

For instance, consider a DSL having *stacks* as a predefined types. The language primitive *pop* is an instance of the pattern described in Figure 13. The operation *pop s* always verifies whether stack s is empty or not. If stack s is empty, then an exception is thrown. An analyzer which infers the size of stack s may be used to avoid the emptiness test. Such analyzer will infer the range of sizes that a given stack may have in a given program point, and it has some similarities to the interval analyzer. Notice also that operation *push* is approximated by operation *+1*, and *pop* is approximated by operation *-1*.

Most high-level programming languages and DSLs save us from managing memory allocations and deallocations. Such languages use garbage collection techniques. However, the use of garbage collectors is often associated with a performance penalty. Shape analysis [JM81] could be applied to make very efficient use of dynamic storage management. For example, such analyzer may infer whether a dynamically allocated memory cell may be converted in a statically allocated one, or whether it may be allocated to a program stack. Once again, a language primitive may be simplified when a given condition is satisfied.

Our last example is related to languages supporting term unification, such as *Prolog*. The unification of terms t and u amounts to solving equations $t = u$ over the (meta) variables in the terms. In the standard situation, all parameters and functions are distinct – the only equalities that hold are reflexivity and the substitution of equals for equals. Thus $f(?a) = b$ [6], $a = b$, and $f(?c) = g(?a, ?b)$ are each unsolvable, while $g(t_1, t_2) = g(u_1, u_2)$ solves $t_1 = u_1$ and $t_2 = u_2$ simultaneously. A term t is said to be *free*, if $t = ?a$. A term t is said to be *ground*, if t does not contain any (meta) variable. The general unification algorithm is computationally expensive. Therefore, we may use an analysis tool that infers the instantiation of a given term in a given program point. For example, consider unification $t = u$, if the analysis tool infers that t is *free*, and u is *ground*, then unification may be substituted by an assignment ($t := u$), which is much more efficient. Such optimization is an instance of another pattern we have identified. Such pattern may be abstractly represented in the following way.

[6] The notation $?a$ is used to represent (meta) variables.

implementation of P
{
  if $(cond_1)$
  { *do* $action_1$ }
  ...
  if $(cond_n)$
  { *do* $action_n$ }
}

Figure 14: Language primitive or user-defined function P

Consider a language primitive or user-defined function P described in Figure 14. If a program analysis tool infers that condition $cond_i$ is always valid in a given point of the program, then a customized version of P may be automatically generated by selecting $action_i$ associated with $cond_i$.

2.3 Verifiers

The verification tools produced by our framework are not intended to prove program correctness. The main objective of such tools is to detect errors. We believe the most important goal of formal methods is to detect the presence of errors. Even when one is trying to prove program correctness, the really interesting cases are the failed proofs, because they hint at errors.

We use state space exploration techniques to detect deadlocks, assertion violations, temporal claims violation [7], etc. Approximations are also used in verification processes, however they may be more precise than the ones used in the analysis algorithms. More precise approximation may be used because complete coverage of the program state space is not necessary, in principle, to detect errors. It is important to keep in mind that analysis tools are only useful, if the generated abstract state spaces reflect *all* possible behaviors (traces) of a given program. As stated above, information produced by an analysis tool is often used to apply optimizations and transformations improving efficiency and quality, and such improvements are only correct if the produced information reflects all possible behaviors of a given program. On the other hand, verification tools are still useful, if they are able to detect a particular trace violating some user stated claim, even if they are not able to prove that such erroneous trace does not exist.

Actually, it is not possible to prove that a program or specification is correct, *i.e.* what we really want. Even when using formal methods, there is no way to guarantee correctness and completeness of a specification with respect to the user's informal requirements. Moreover, true user requirements might be different from what the users say they need, and will often vary with time. At the most, we can prove that a given specification has some properties, or a given program satisfies a given specification. Therefore, we believe that software development is a continuous process of increasing our confidence that a program/specification is "correct".

[7] Temporal claims are specified in linear temporal logic.

Our point of view is analogous to the *principle of plausible reasoning* described by Polya [Pol54]: "a conjecture becomes more credible by the verification of any new consequence". In other words, if we have "$A \Rightarrow B$" and "B is verified (or proved)", then "A becomes more credible". Such principle is constantly applied in the natural sciences, and it may be mapped to computer science as: if "program/specification A should have property B" and "B is verified or its absence is not detected", then "A becomes more credible", *i.e.* the developers increased their confidence that the program/specification is "correct".

Executing the program/specification in particular cases is the simplest way of increasing our confidence that a software is correct. It is the most used technique in industry. Unfortunately, such technique provides poor results when applied to concurrent systems. The main problem is that only a small part of the state space is usually covered, and scenarios leading to errors are hard to reproduce. Anyway, the simulation tools described above may be used to perform this simple verification technique.

Most programs have potentially infinite state space, whether they are coded by using DSLs or not. However, if the abstract state space is finite structure due to the use of semantic approximations, then we might also use a two-step algorithm to perform program verification. Such algorithm is almost analogous to the one used in program analysis. The main difference is the *purpose* of the algorithm. Analysis algorithms *collect* information, *i.e.* they *infer* program properties. On the other hand, verification algorithms *check* properties, and *produce* counter-examples when a given property is not valid. However, the two-step verification algorithm is feasible only for very simple programs, once a *reasonable* finite abstract state space will often be too large to be stored in memory. Notice also that if too much approximation is applied to reduce the size of abstract state space, then several unrealistic behaviors will also be introduced. To sum up such problem, we have that:

1. Precise abstract state spaces are too *large* or infinite data structures.

2. Imprecise abstract state spaces are unrealistic for verification purposes.

It should be clear that the use of extremely imprecise abstract state spaces is not a problem in the program analysis context. At most, imprecise safe properties will be inferred, and some program optimizations and transformations will not be allowed.

2.3.1 Verifying large and potentially infinite state spaces

To cope with the problems described above, we have defined several verification algorithms that "interpret" and perform verification at the same time. In this approach, we do not need to explicitly store the (abstract) state space. Figure 15 contains an algorithm that abstractly represents all verification algorithms of our framework. It may seen redundant, since it is representing different verification techniques. Variable *worklist* is a list of pairs $\langle$*state*, *curr-trace*$\rangle$, where *state* is a state to be visited, and *curr-trace* the associated *current trace*. Variable *cache* contains a cache of already visited states, but not necessarily all of them. Function *add(cache, curr-state)* inserts state *curr-state* in the specified cache. Function *divergence-detection* verifies whether the current trace is diverging or not. It should be clear that only one of the first two statements *if* is necessary,

```
worklist = cons(⟨s0, empty-list ⟩, empty-list);
cache = empty-cache;
while (worklist ≠ empty-list)
{
  ⟨ curr-state, curr-trace ⟩ = remove-first(worklist);
  curr-trace = cons(curr-state, curr-trace);
  add(cache, curr-state);
  for each succ-state in successors(curr-state)
  {
    if (is-in-cache(cache, succ-state))
    { // state was already visited!
      // stop the verification of curr-trace
    }
    else if (is-in(succ-state, curr-trace))
    { // cycle detected!
      // stop the verification of curr-trace
    }
    else if (divergence-detection(curr-trace))
    { // current trace is probably diverging!
      // stop the verification of curr-trace
    }
    else
    { // add succ-state to the worklist!
      worklist = cons(⟨ succ-state, curr-path ⟩, worklist);
    }
  }
}
```

Figure 15: Simple depth-first search verification algorithm

if variable *cache* contains at least the states in the current trace. The second statement *if* should be considered only when *caches* are not used, *i.e.* statements related to variable *cache* are removed from the algorithm.

This simple verification algorithm is similar to the analysis algorithm described before (Figure 5), in case we make variable *cache* stores *all* visited states, and (abstract) state space is finite. *Divergence detection* is not necessary, once the state space is finite.

Bit-state exploration (supertrace) used in the system *Spin* [Hol97] may be also seen as an instance of our verification algorithm. In bit-state exploration, a hashcode is computed for each already visited state. Such hascode is used to set the associated bit in a bitmap (array of bits). The size of such bitmap is very important, once whenever a system state is analyzed by comparing its hashcode with an already existing values a collision risk exists. In this case, variable *cache* refers to an array of bits, function *add(cache, state)* is implemented as *cache[hash(state)] = 1*, where the function *hash* computes state's hashcode. Function *is-in-cache(cache, state)* is implemented as *cache[hash(state)] == 1*.

Bit-state exploration ensures termination even if state space is infinite, once at the most N states may be visited, where N is the size of the array of bits. Unfortunately, the definition of function *hash* is the only way to control the slice of the state space which will be covered in the verification process. For instance, we have no guarantee of how deep a particular infinite trace has been explored. Therefore, we have defined the concept of divergence detection functions. Such functions allows us to have total control over how the verification of current trace is interrupted due to divergence problems. Divergence detection functions are particularly interesting when we are analyzing programs containing infinite traces $(s_0, \ldots, s_n, \ldots)$, where $s_i \neq s_j$ for every $i \neq j$, *i.e.* traces which does not contain cycles. For example, consider the following divergence detection functions:

Number of steps The simplest divergence detection function just counts the number of states in the current trace, and verifies whether it is greater than a predefined value.

$$f_{max}(trace) \stackrel{\text{def}}{=} (length(trace) > numMaxSteps)$$

Hashing Bit-state exploration may be adapted as:

$$f_{hash}(trace) \stackrel{\text{def}}{=} \exists s_1, s_2 \in trace \cdot hash(s_1) = hash(s_2)$$

It is clear that our version is more precise than standard supertrace, since our version will not visit a state, if there is another state with the same hashcode in the *current trace*. Instead the standard supertrace algorithm will visit whole state space. A more precise version of f_{hash} is:

$$f_{hash}(trace) \stackrel{\text{def}}{=} \exists s_1, \ldots, s_n \in trace \cdot hash(s_1) = \ldots = hash(s_n)$$

In such case, current trace is aborted if there are at least n states with the same hashcode.

Weight Consider a function *weight* that returns the *divergence potential* of a given state. So a divergence detection function may be defined as:

$$f_{weight}(trace) \stackrel{\text{def}}{=} \exists s_1, \ldots, s_n \in trace \cdot weight(s_1) < \ldots < weight(s_n)$$

In other words, f_{weight} detects divergence if there is a subsequence of length n, where the *divergence potential* is increasing. k

Conjunction If f_1 and f_2 are divergence detection functions, then

$$f_{\wedge}(trace) \stackrel{\text{def}}{=} f_1(trace) \wedge f_2(trace)$$

is a divergence detection function too, and it is more precise than f_1 and f_2 [8].

[8] f is more precise than f', if and only if, $\forall trace \cdot f(trace) \Rightarrow f'(trace)$. When f erroneously detects a trace as divergent, so does f'.

2.3.2 Checking assertions

Assertions are the simplest properties that may be verified in a program. Briefly, when a verification tool "finds" an assertion *assert(P)*, it simply checks whether a condition P is valid in the current state (*curr-state*). To implement this verification in the algorithm in Figure 15, we should insert the "test" after statement ⟨ *curr-state, curr-trace* ⟩ = *remove-first(worklist);*.

Now consider property *Assert(P)* that checks if all direct and indirect successors s_j of current state s_i ($s_i \twoheadrightarrow s_j$) satisfy condition P. The verification of this kind of property is also easy to be implemented. We should transform tuple $\langle state, curr\text{-}trace\rangle$ in $\langle state, curr\text{-}trace, conds\rangle$, where *conds* is a list of conditions to be verified. For every reached state, all conditions in the list *conds* are verified. When a property *Assert(P)* is found by the verification tool, P is inserted in the list *conds*.

2.3.3 Simple *deadlock* and *livelock* detection

Deadlocks may be easily detected. We just need to check whether there exists an enabled transition for the current state ($successors(curr - state) = \emptyset$), and if the program is not in a valid terminal state. Livelocks may be also easily detected, if we use an approximation. Briefly, the user should decorate the program with *progress* marks. So we say the program is in *livelock* if a progress mark is not reached after a user-defined number n of consecutive transitions. It is clear that such approach may signal "false" errors, once n may not be big enough. However, it is efficient and easy to implement.

2.3.4 Temporal properties

To perform verification of temporal claims, *i.e.* formulas described in linear temporal logic, we use an approach similar to the one used in SPIN [Hol97]. The temporal logic formula is converted into a Büchi automaton, and the synchronous product of this claim and the state space is computed. In other words, we only need to transform tuple $\langle state, curr\text{-}trace\rangle$ into $\langle (state, buchi\text{-}state), curr\text{-}trace\rangle$, where *buchi-state* is a state of the Büchi automaton associated with the linear temporal logic formula. Notice also that such technique is compatible with the divergence detection functions described in the previous section.

2.3.5 Assure Marks

As we have discussed, the use of approximations implies in undesirable behaviors. Thus, a verification tool may report "false" errors. We cope with such problem by using *assure marks.* Such marks are similar to assert marks. However, when the condition associated with an assure mark is not satisfied by the current state, the verification tool discards the *current trace*, instead of generating an error message. In other words, verification tools always assume that conditions associated with assure marks are always true, thus if the condition is not satisfied by the current state, such state should not be a "real" state, *i.e.* it was reached due to the use of approximations. Assure marks allow us to minimize the imprecisions associated with the use of approximations.

Assure marks may be also used in analysis tools. The purpose here is to improve the analysis quality, *i.e.* to infer more precise program properties. Some compilers already use some kind of marks which may be regarded as instances of our assures marks. The compiler of the lazy function language *Clean* [PE93] is an example. The Clean compiler allows the user to specify whether a function is *strict* in a particular argument or not [9]. It is well known that eager evaluation is more efficient than lazy evaluation. Thus, it is often profitable to convert lazy evaluation into eager evaluation. Such transformation is safe if and only if the function is strict in the particular argument. The user's strict annotations are used to improve the results of Clean's strict analysis.

We should be careful when using assure marks, once if we state a "false" condition, we shall be hiding several errors. In the context of analysis tools the problem is even worse, since in such case "false" conditions may allow invalid transformations or optimizations. For example, in the Clean compiler, if we erroneously state that a parameter is strict, a terminating program may be transformed into a non-terminating one.

The analogy between *assert* and *assure* may be extended to temporal claims. In this case, the verifier works in a way similar to the one described in the previous section. Nevertheless, the temporal property describes a situation that always holds true, thus the verification tools discard the current trace, if the property is violated. Notice that such use of assure marks has some similarity with the *fairness conditions* used in standard model checkers.

2.3.6 Additional Issues

Partial Order Methods [God96] are used to tackle the "state space explosion" phenomenon due to interleaving. Unfortunately, it is the user responsibility to specify the notion of independent transitions for a given language. However, if we are reusing fragments of semantic descriptions, we can also reuse the specified notion of independent transitions. State compression techniques [Hol97] are also responsibility of the user. We provide, at the most, semantic descriptions fragments which use *store* structures that reduce memory consumption.

2.3.7 Verification Examples

Instead of describing well known examples related to concurrent languages, we focus on unusual applications of the verification techniques described above. Our intention is to illustrate that such techniques may be used to detect programming errors such as invalid array indices, null pointer dereferencing and other common errors.

Figure 16 contains a simple function that contains several errors. This example is based on an example used to exemplify system ESC [DLNS98]. ESC is a verification tool based on theorem proving. Function *arrayCat* concatenates the elements of string array a. Abstract data type *StringWriter* is an output stream. Function *putString* "puts" a given string in the end of the stream. Function *getString* returns the stream's contents. To verify this program, we will use the following semantic approximations:

[9] A function f is strict in a particular argument if $f(\perp) = \perp$, where $\perp$ means a non-terminating computation. The intuition is that if a function is strict in a particular argument then that argument is definitely required to compute f.

```
string arrayCat(string [ ] a)
{
  wr = new StringWriter();
  for (i = 1; i <= size(a); i++)
    putString(wr, a[i]);
  return getString(wr);
}
```

Figure 16: Function *ArrayCat*

1. *String* values are approximated by abstract values *null* and *non-null*.
2. *StringWriter* values are approximated by abstract values *uninit* and *init*, representing a *StringWriter* value before and after initialization.

Now, we may perform verification considering the contexts of use of function *arrayCat* in the whole program. Another possibility is to write an auxiliary function that artificially produces different contexts of use. Anyway, the verification tool shall detect the following errors:

1. Function *putString* is applied over an uninitialized object (uninit). The user may fix such problem by adding statement *initStringWriter(wr)*
2. Invalid use of function *putString*, the string *a[i]* may be *null*. The user may fix such problem by adding a *if* statement such as:

   ```
   if (a[i] != null)
     putString(wr, a[i]);
   ```

3. Invalid array index in expression *a[i]*. The user may fix such problem by exchanging

   ```
   for (i = 1; i <= size(a); i++)
   ```

   for

   ```
   for (i = 0; i <= size(a) - 1; i++)
   ```

Figure 17 contains another example, where the following approximations are used:

1. Integer values are approximated by ranges.
2. File handlers are approximated by abstract values *closed* and *opened*.

If the maximum value of the range associated with variable i is greater than 1023 at the first *read* statement, then an invalid array index error is detected. If the error was detected due to the use of semantic approximations, then the user may add *assure*($i < 1024$) to avoid the generation of spurious error messages. The verification tool will also report an error in the last *read* statement, once the abstract value associated with *fd* is *closed*.

```
int i;
char buf[1024];
file fd; // declares a file handler
...
fd = open(filename);
...
read(fd, buf, i); // read i bytes, and copy them to buf
...
close(fd);
...
read(fd, buf, i);
```

Figure 17: Program *file access*

3 Framework implementation issues

So far, we assumed that the operational semantics of a language defines a function $successors : State \to \wp(State)$, *i.e.* a function that given a state returns all possible successors of such state. Such abstract point of view was sufficient to describe our analysis and verification algorithms. However, the key point of the framework implementation is how the user describes the operational semantics of a given language. Therefore, we allow the user to provide *Structed Operational Semantics*[Plo81] (SOS) descriptions for the desired language. The user is also encouraged to provide modular SOS descriptions in order to allow the reuse of simulation, analysis and verification tools.

The implementation of the framework involves the use of a new logic programming language called PAN [MLH00a] which was specially developed to implement analysis and verification tools. As we use a modular approach for SOS, the user can reuse fragments of semantic descriptions. As mentioned before, the framework contains different algorithms to perform simulation, analysis and verification. We also provide a library of semantic approximations that is used to simplify the process of defining new program analyzers.

In the following subsections we show a short description of modular SOS, and an introduction to PAN. The main objective is to describe how an efficient implementation was achieved.

3.1 Modular Structural Operational Semantics

In this section we provide a brief introduction to the method of *Structural Operational Semantics* (SOS). SOS represents computations by means of deductive systems, transforming an abstract machine into a system of logical inferences. Since the semantic descriptions are codified in a deductive way, proofs about program properties can be directly derived from the definition of language constructors, using structural induction.

SOS definitions are given by inference rules consisting of conclusions that follow from a set of premises. Such rules describe possible transitions (*small steps*) of a computation. Each proof in the deductive system describes a state transition.

SOS involves abstract syntax, computed values, configurations and inference rules for (labelled) transitions. A SOS description specifies a *labelled transition system* $(\Gamma, T, A, \rightarrow)$, where Γ is a set of configurations, $T \subseteq \Gamma$ is a set of terminal configurations, A is a set of labels, and $\rightarrow \subseteq \Gamma \times A \times \Gamma$ is a transition relation. For configuration $\gamma, \gamma' \in \Gamma$ and labels $\alpha \in A$, the assertion that $(\gamma, \alpha, \gamma')$ is in the transition relation is written as $\gamma \xrightarrow{\alpha} \gamma'$. It is clear that transition relation $\rightarrow$ defines the function *successors* described before.

On the other hand, for our objectives SOS descriptions can be seen as interpreters implemented in a logic programming language. The main predicate of these logical programs is the one which defines the transition relation.

However, most of the SOS descriptions lack modularity. Local modifications in language semantics usually imply global modifications in SOS descriptions. This also means that it is almost impossible to *reuse* SOS descriptions when defining a new language. To cope with this problem, we use an approach similar to the one defined in [Mos99]. In this approach, the modularity is achieved by using encapsulation techniques. Figure 18 shows a small fragment of a modular SOS description for a simple imperative language. In this example, the configurations are restricted to abstract syntax trees, where nodes may be replaced by the values they computed, as in conventional SOS. The initial configurations are pure syntax, and terminal configurations are simply computed values. Value *completed* represents normal termination. All usual semantic components of configurations, such as stores, are incorporated into the labels on transitions. All labels contain a pair of stores reflecting the value of the store before and after the transition. The structure of the label is encapsulated, and should only be accessed by means of predicates. We can view the label as an object [10] and the predicates as the interface (methods) of the object. Notice that the computation state before a particular transition is defined by the abstract syntax tree plus label components that reflect the state before the transition (e.g. the store before the transition). As we will see later, our program tools are all based on the concept of computational state.

All labels contain a partial composition predicate used to assert whether two transitions may be adjacent. In the example in Figure 18, this predicate is defined as:

$$compose(\alpha_1, \alpha_2, \alpha) \stackrel{\text{def}}{=} \begin{cases} \alpha_1 & = & \langle store_1, store_2 \rangle \wedge \\ \alpha_2 & = & \langle store_2, store_3 \rangle \wedge \\ \alpha & = & \langle store_1, store_3 \rangle \end{cases}$$

This predicate asserts that label α is the composition of labels α_1 and α_2. Every label can always be composed on the left and on the right with a specific identity label. The notion of identity labels is defined by the predicate *id*. In the given example, this predicate is defined as:

$$id(\alpha) \stackrel{\text{def}}{=} (\alpha = \langle store, store \rangle)$$

We use $\gamma \xrightarrow{i} \gamma'$ as a syntax sugar for:

$$\frac{id(\alpha)}{\gamma \xrightarrow{\alpha} \gamma'}$$

[10] Using the object-oriented terminology.

$$skip \xrightarrow{i} completed \qquad \frac{set\text{-}store(\alpha, Id, Val)}{Id := Val \xrightarrow{\alpha} completed}$$

$$\frac{Be \xrightarrow{\alpha} Be'}{if\ Be\ then\ S_1\ else\ S_2 \xrightarrow{\alpha} if\ Be'\ then\ S_1\ else\ S_2}$$

$$if\ true\ then\ S_1\ else\ S_2 \xrightarrow{i} S_1 \qquad if\ false\ then\ S_1\ else\ S_2 \xrightarrow{i} S_2$$

$$while\ E\ do\ S \xrightarrow{i} if\ E\ then\ S; while\ E\ do\ S\ else\ skip$$

$$\frac{S_1 \xrightarrow{\alpha} S_1'}{S_1; S_2 \xrightarrow{\alpha} S_1'; S_2} \qquad completed\ ; S_2 \xrightarrow{i} S_2$$

Figure 18: Modular SOS example

As already mentioned, all labels should provide predicates to access their internal structure. In the given example, the label contains two predicates:

$$\begin{aligned} set\text{-}store(\alpha, Id, Val) &\stackrel{\text{def}}{=} \alpha = \langle S_1, S_2 \rangle \wedge\ S_2 = S_1[Id \leftarrow Val] \\ get\text{-}store(\alpha, Id, Val) &\stackrel{\text{def}}{=} \alpha = \langle S_1, S_2 \rangle \wedge\ Val = S_1(Id) \end{aligned}$$

The predicate *set-store* "modifies" the value of a given variable in the store. The predicate *get-store* "accesses" the value of a given variable.

Using this approach, we do not need to completely modify a SOS description when we extend or modify the language semantics. The use of labels is not the only way to modularize SOS descriptions. We can also apply the encapsulation approach to transitions of the form $\gamma \rightarrow \gamma'$, where configurations γ and γ' are composed of terms and semantic components as stores and environments [11]. Here, the main idea is to encapsulate the structure of the semantic components. Therefore, we have $\gamma = \langle term, \Sigma \rangle$, where Σ is the structure which encapsulates semantic components. Once again, predicates are used to access the internal structure of Σ. Using this modeling approach, the rules defining the statements *skip* and *assignments* are:

$$\frac{id(\Sigma_1, \Sigma_2)}{\langle skip, \Sigma_1 \rangle \rightarrow \langle completed, \Sigma_2 \rangle} \quad \frac{set\text{-}store(\Sigma_1, Id, Val, \Sigma_2)}{\langle Id := Val, \Sigma_1 \rangle \rightarrow \langle completed, \Sigma_2 \rangle}$$

Observe that in this approach, the computation state is defined as $\langle term, \Sigma \rangle$. This way, state space exploration tools that use *state caches* can be easily defined.

Of course, it is also possible to use hybrid approaches with transitions such as $\langle t, \Sigma \rangle \xrightarrow{\alpha} \langle t', \Sigma' \rangle$. Independently of the structure of the transitions, the desired modularity is achieved when the structure of the semantic components is not directly accessed.

SOS descriptions usually do not require implementation *tricks* such as using *stacks* to implement local variables. However, we do not see any problem in using

[11] This approach is due to C. Braga.

such low-level structures in the language semantics, providing they improve the performance of analysis and verification tools, or simplifies the definition of semantic approximations.

As noted by Schmidt [Sch98], SOS definitions can generate new programs on the fly – arbitrary SOS derivations are not semicompositional and this can hinder the termination of our analysis tools. The proposed technique was to abstract the syntax of the source language. The use of low-level structures in the semantics also avoids the presence of non semicomposicional SOS derivations.

Further information on writing modular semantic specifications, and reusing fragments of analysis and verification tools is found in [MLH00b].

3.2 The *PAN* Specification Language

In our framework, the operational semantics of a language must be specified in a programming language called *PAN*. PAN is a logic programming language specially developed to specify interpreters, program analyzers and verifiers. The PAN syntax and semantics have some similarities with *Prolog* [SS94]. The PAN compiler uses global analysis to produce code with a performance comparable to the one produced by imperative language compliers.

All PAN programs must be properly qualified by providing the kind, type and mixfix declarations. For instance, consider the following specification of the concatenation of lists in PAN syntax:

```
append nil X X.
append (X :: Xs) Ys (X :: Zs) :- append Xs Ys Zs.
```

This specification contains four variables `X`, `Xs`, `Ys` and `Zs`, one logical constant, namely `:-` for the converse of implication, and three non-logical constants, namely `nil`, `::` and `append`, denoting the empty list, the list constructor, and the concatenation relation, respectively. Two of these constants, namely `:-` and `::` are also used as infix symbols. This specification is only meaningful if mixfix declarations and type declarations are given for these constants.

```
type nil : list A.
type _ :: _ : A -> list A -> list A {prec 8 r-assoc}.
type append : list A -> list A -> list A -> o.
```

PAN has a polymorphic type system similar to the type system of ML [Mil78]. In type declarations, tokens with an initial upper case letter denote *type variables*. The type declaration for `nil` above asserts that for every type `A`, the constant `nil` is of type `(list A)`. The type declaration for `_ :: _` asserts that given a term `H` of type `A`, and a term `T` of type `list A`, then the term `T :: H` is of type `list A`. If the target type of a type declaration is `o`, then the constant is a predicate symbol. In the above example, `append` is a predicate symbol that contains three arguments.

The underbars in constants like `_ :: _` are place holders that indicate where terms of type `A` and `list A` should go, respectively. The precedence attribute `prec` n tells how tightly a symbol binds. The higher the number the tighter the symbol binds. The attributes `r-assoc` and `l-assoc` tell if a symbol is right or left associative. Precedence and associativity information is used to disambiguate the parsing. Notice that the above declarations have introduced two new constants not in the original specification, namely `o` and `list`. These type constants will need declarations, called *kind declarations.* Kind declarations are used to introduce *type constructors.* An example of kind declaration is:

```
kind list : type -> type.
```

Here, `list` takes a type and returns another type. For example, `list (list int)` is the type of a list of lists of integers.

PAN also supports higher orders predicates, i.e. predicates that have other predicates as arguments. For example:

```
type map : (A -> B -> o) -> list A -> list B -> o.
map P nil nil.
map P (X :: Xs) (Y :: Ys) :- (P X Y), map P Xs Ys.
```

The goal `(map F Xs Ys)` is provable, if `Xs` and `Ys` are lists of equal length and corresponding elements of these lists are related by the predicate `P`.

Prolog has various all-solutions predicates (e.g. findall/3), all of them having semantic problems. PAN has the builtin predicate called *solutions.* We avoid the variable scoping problems of most Prolog versions. Rather than taking a goal to execute and an aliased term holding the resulting value to collect, the predicate *solutions* take as input a single higher-order predicate. The declaration of the predicate *solutions* is:

```
type solutions _ _  : ((A0 -> o) -> (list A0) -> o)
```

Such predicate is used in the implementation of analysis and verification algorithms. In fact, the predicate *solutions* is used to obtain all enabled transitions of the current state.

PAN compiler is based on state of the art logic programming compilation technology [RD92, SHC96]. The compiler uses several global analysis (mode analysis, determinism analysis, switch detection, etc) to produce efficient code. It is out of scope of this article to explain each one of such analysis algorithms. The interested reader should refer [MLH00a] . The output of the PAN compiler is portable C code. Hence, the PAN compiler allows us to invoke C code from PAN code and vice-versa. This feature is *fundamental* for an efficient implementation of our analysis and verification algorithms.

PAN does not have "impure" features like Prolog [12]. At first, this looks like extremely contradictory, once we dared to use a pure logic language to produce efficient implementations of tools usually implemented by imperative languages. We avoid the efficiency problems by using an approach similar to the one found in pure functional languages [Wad90]. Briefly, the user can decorate the code with uniqueness marks. Such marks tell the compiler that a given argument of a given predicate can only have one variable referencing it. As there is only one reference to a given argument, destructive updates can be safely performed. This safety condition is checked by the compiler, and if not satisfied produces a warning message. This feature is used to allow PAN code to invoke C code which performs destructive updates in a safe way. Figure 19 shows an example of interface between PAN and C code. The first three lines show how to instruct the compiler to add arbitrary C code into the produced

[12]The only non-logical feature of the PAN language is the *cut.*

```
C{{
#include "array.h"
}}C.
kind array : type -> type.

cdecl create-empty-array : unq out array A -> o {det}
C{{ // C code ... }}C.

cdecl update-array : in int -> in A -> unq in array A ->
                     unq out array A -> o {det}
C{{ // C code ... }}C.
```

Figure 19: Interface with the C language

```
kind stmt : type.
kind expr : type.

type _ ; _ : stmt -> stmt -> stmt {prec 80 l-assoc}.
type skip : stmt.
type if _ then _ else _ end :
                 expr -> stmt -> stmt -> stmt {prec 65}.
type while _ do _ end : expr -> stmt -> stmt {prec 70}.
type _ := _ : id -> expr -> stmt {prec 100}.
```

Figure 20: Abstract syntax definition

code. The *cdecl* declarations tell the compiler that the respective predicates are implemented by using C code. The marks *in* and *out* are used to indicate which arguments are input or output. The uniqueness marks are represented by the symbol *unq*. For instance, the predicate *update-array* receives an index, a value and the array that will be updated. The resulting array is returned in the fourth argument.

The PAN features are extremely useful for specifying the operational semantics of a programming language. The mixfix notation allows us to specify the abstract syntax of a programming language and to describe the transition rules in a clear way. For instance, Figure 20 shows part of the abstract syntax definition of the simple imperative language.

Figure 21 shows part of the translation to PAN of the SOS description of Figure 18. The symbol `|-` is an alternative for `:-` that simplifies the description of SOS rules. It allows us to write a clause as `Body |- Head` instead of `Head :- Body`. The predicate `_ -- _ --> _` specifies the transition relation.

```
kind label : type.
kind config : type.
sub stmt : config.
type completed : config.
type _ -- _ --> _ : config -> label -> config -> o.

S1 -- L --> S1'
|-
S1 ; S2 -- L --> S1' ; S2.

id L
|-
while E do S end -- L --> if E then S ; (while E do S end) end.

(is-val E V), (set-store L ID V)
|-
ID := E -- L --> completed.
```

Figure 21: Fragment of language semantics

3.3 Partial evaluation

In the previous section, we mentioned that encapsulation techniques are fundamental to achieve modular SOS descriptions. Unfortunately, the use of encapsulation techniques are often associated with a degradation of execution performance. For instance, calling a general predicate instead of performing the actions directly in-line generally results in slower executions. However, partial evaluators [JGS93] open new possibilities for writing modular code without sacrificing performance. We are currently developing an automatic partial evaluator for the PAN language based on the ideas used to implement *Mixtus* [13] [Sah91]. The absence of "impure" features in the PAN language avoids several problems found in the implementation of Mixtus.

Partial evaluation is a program transformation which specializes a program to a particular context reducing its execution time and, in some cases, its size [JGS93]. A specialization context is defined by assigning values to some subset of a program's inputs. More specifically, consider a program p, taking arguments s and d and producing result r, which is written as $p(s,d) = r$. If s is a known subset of the input, which describes the context, and d is yet unknown, we may form a new program p_s, where computations depending only on s have been performed, and $p_s(d) = p(s,d)$. The program p_s is called a specialization of p with respect to the known input s. A partial evaluator is a program that computes p_s.

Futamura [Fut71] showed that a partial evaluator applied to an interpreter (for language X, written in a language Y) given a program will yield a new program (in language Y). This is called the first Futamura projection, and is probably the most useful and widely used aspect of partial evaluation, as the

[13] Mixtus is an automatic partial evaluator for Prolog.

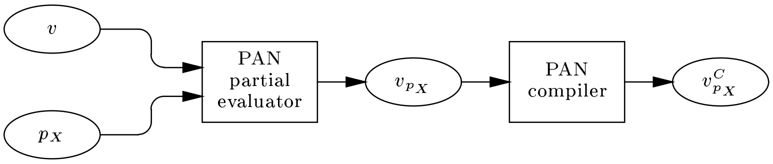

Figure 22: Partial Evaluation

overhead of the interpreter might be eliminated. The first Futamura projection shall be widely applied in our tools, once all of them may be seen as special kinds of interpreters. More specifically, consider a verification tool v for a language X. As described above, such tools are implemented using *PAN*. The inputs of tool v are a program p_X written in language X, and an initial state s_0. Now, using partial evaluation, we obtain program v_{p_X} which is a specialization of v with respect to the program p_X. Notice that, v_{p_X} is a PAN program intended to verify program p_X. The PAN compiler converts program v_{p_X} in an equivalent C program $v^C_{p_X}$. Figure 22 summarizes this process. Needless to say, a C compiler must be used to convert $v^C_{p_X}$ in executable code.

It is only fruitful to apply the above process to generate optimized versions of complex analyzers and verifiers, once partial evaluation execution time may be greater than the execution time of simple analyzers and verifiers. In general, it is always profitable to apply the partial evaluation technique in verification procedures, once they may take hours to be completed.

Independently of the above process, partial evaluation may also be used to avoid the performance problems associated with encapsulation techniques used in our semantic descriptions. In this case, we are transforming the tool v in a tool v', without specifying any input. Such transformation shall improve the performance, because it will *unfold* several predicates (procedures), opening, this way, new optimization possibilities. It is important to remember that when we are reusing a semantic module, it is common to *fix* the value of arguments of some predicates, thus partial evaluation is used to specialize a reusable semantic module.

4 Using the framework

So far, we have described several examples of analysis and verification. In this section, we will focus on pragmatics issues related to the use of our framework. We describe a realistic application of our framework in the domain of industrial process control and programmable logic controllers (PLCs) [AT98].

4.1 DSL description

PLCs can be programmed to compute a particular, usually time-varying relation between a set of controlled outputs and a set of process-related inputs. In 1993, the International Electrotechnical Commission (IEC) published the IEC 1131 International Standard for Programmable Controllers. Part 3 of this standard [Lew95] defines a suite of four domain-specific languages. Similar to [AT98], we are only concerned with the subset of the language referred to as *Sequential Function Charts* (SFCs). These are programming elements inspired by Petri

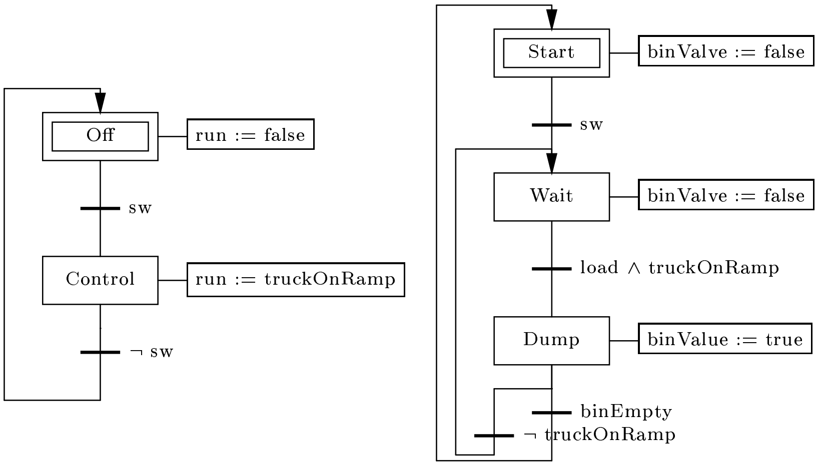

Figure 23: Gravel SFC

nets and are used to specify how the overall input-output relation is computed by the PLC – *i.e.* the mode of the system – changes in response of internal and external events.

We exemplify language SFC by using the same example described in [AT98]. In such example, the task is to transfer a measured amount of gravel from a bin onto a truck under human supervision. Human control is performed through a panel of two on/off switches which the status is represented as boolean variables *sw* and *load*. The other two inputs to the controller come from two sensors represented as boolean variables *binEmpty* and *truckOnRamp*. Based on these inputs the controller computes boolean outputs *run* and *binValue* to control the motion of the conveyor belt and the valve at the bottom of the bin.

SFCs are usually represented diagrammatically as in Figure 23. Nodes are of two distinct kinds: rectangular boxes called steps and thick horizontal lines called transitions. Each step has a single associated action which is displayed an oblong attached box. Transitions are labelled by boolean expressions and are said to be enabled when their corresponding expression evaluates to *true*. A number of steps may be designed as *initial*; these are distinguished by a double border.

The program in Figure 23 is composed of two simple SFCs in parallel. The SFC on the left handles input from the control panel and runs the conveyor belt, the one on the right controls the bin.

We use the same textual syntax described in [AT98] for the expression of SFC diagrams. The graphical representation may be automatically extracted from the textual representation. The formal syntax is described as:

$$P ::= \langle a \rangle \mid P \ ; \ G \mid P_1 \parallel P_2 \mid \textbf{loop } P : b$$

$$G ::= b \to P \mid G_1 [\!] G_2$$

P ranges over programs, and G over guarded programs. The term $\langle a \rangle$ represents a single step with associated action a, b stands for the condition for performing a transition. $P \ ; \ G$ is sequential composition via the transition in G, $P_1 \parallel P_2$ is parallel composition, **loop** $P : b$ is a loop with body P and transition b along the link to the top of the loop, and $G_1 [] G_2$ is the nondeterministic choice between branches G_1 and G_2. The diagram of Figure 23 is described as:

$$
\begin{array}{l}
\mathbf{loop}(\langle \mathit{run} := \mathit{false} \rangle \ ; \ \mathit{sw} \to \langle \mathit{run} := \mathit{truckOnRamp} \rangle) : \neg \mathit{sw} \\
\| \\
\mathbf{loop}(\langle \mathit{binValve} := \mathit{false} \rangle \ ; \\
\qquad \mathit{sw} \to \mathbf{loop}(\langle \mathit{binValve} := \mathit{false} \rangle \ ; \\
\qquad\qquad\qquad (\mathit{load} \wedge \mathit{truckOnRamp} \to \langle \mathit{binValve} := \mathit{true} \rangle) \\
\qquad\qquad\quad) : \neg \mathit{truckOnRamp} \\
\quad) : \mathit{binEmpty}
\end{array}
$$

The actions a and conditions b are described using the simple imperative language described in Section 3.1. Thus, we may reuse the semantic modules describing such simple language in the definition of SFCs. It should be clear that other languages, such as ladder logic, could have been used to describe actions and conditions, once their semantics had been specified using PAN.

Notice that the operating environment of a PLC is assumed to produce signal changes at a rate which is much slower than the speed of the PLC itself. Thus, we may assume that interaction with the environment is *synchronous*, *i.e.* changes only happen at particular, discrete instances determined by clock of the PLC system.

4.2 Parsing

The first step in producing analysis and verification tools is to implement a parser for the desired language. A parser can be generated by a tool such as YACC or the parser generator of the DMS system. The interface between PAN and C allows us to produce a term which could be interpreted by PAN code. Such term is the abstract syntax of the program. Figure 24 contains the specification of the abstract syntax of SFC diagrams in the PAN language. Following that, we specify the semantics of the language based on the abstract syntax. However, we usually simplify the abstract syntax by removing existing syntax sugars in the language, and transforming complex concepts in a collection of simpler ones. The simplification process can be implemented by a transformational system such as DMS, or even using PAN. It is well known that logic programming languages are extremely useful for applying programming transformations [SS94]. Such transformational process usually simplifies the description of language semantics. This practice coincides with the one used in the compiler literature, where all analysis are performed over an intermediate representation.

Several tools produced in our framework should provide some kind of traceability, i.e. the tool must associate the abstract syntax tree (or the intermediate representation) with the original source code. This is specially important if we desire to interact with the tool. Thus, we provide mappings used to associate abstract syntax tree nodes with source code information such as line and column numbers.

```
type < _ > : action -> sfc.
type _ ; _ : sfc -> sfc -> sfc {prec 80 l-assoc}.
type _ || _ : sfc -> sfc -> sfc {prec 75 l-assoc}.
type loop _  _ : sfc -> expression -> sfc {prec 65}.
type _ -> _ : expression -> sfc -> sfc {prec 60}.
type _ [] _ : sfc -> sfc -> sfc {prec 70 l-assoc}.
```

Figure 24: SFC abstract syntax described in PAN

4.3 Language Semantics

Figure 25 contains the semantics of SFC programs. Notice that the semantics depends on the definition of predicate `_ -- _ -->> _` which is the reflexive and transitive closure of the predicate defining the semantics of *actions* and *expressions*. It is clear that the semantics of SFC programs is completely decoupled of the label structure! Therefore, such semantic description may be *reused* even if different languages, such as ladder logic, are used to specify *actions* and *expressions*.

A predefined reusable module *verificationSupport* may be imported. This module defines assertions, progress marks, linear temporal logic formulas, and assure marks, allowing the user to decorate SFC programs with such verification information.

The above example demonstrates that our framework allows the user to combine different domain-specific languages in a easier way, once the structure of the label or state is encapsulated. Such possibility allows us to experiment with new languages combinations. For example, combining SFC with functional languages.

4.4 Generating Tools

As stated before, the language semantics defines the transition relation that is used to build different tools. However, analyzers request the user to specify the desired semantic approximations. SFC is an extremely simple language and most part of the approximations are associated with the language used to describe actions and expressions. The only approximation we may worry about is related with expressions, once the value of an expression might not be precise, *i.e.* a boolean value may be equal to $\top$ (top). Therefore, we should add an extra transition reflecting such case. This transition is defined as:

```
B -- L -->> top
|-
(B -> P) -- L --> P.
```

As our analysis algorithms may be applied to concurrent languages such as SFC, thus standard analysis such as constant propagation, common-subexpression elimination and live variables may be applied to allow the generation of more efficient and compact target code.

Simulations and verifications may be also performed. However, if the user is interested in performing verification, then he/she must model the "surrounding"

```
A -- L -->> completed
|-
< A > -- L --> completed.

P -- L --> P'
|-
(P ; G) -- L --> (P' ; G).

id L |- (completed ; P ) -- L --> P.

P1 -- L --> P1'
|-
(P1 || P2) -- L --> (P1' || P2).

P2 -- L --> P2'
|-
(P1 || P2) -- L --> (P1 || P2')

id L |- (completed || completed) -- L --> completed.

id L |- (loop P B) -- L --> (P ; B -> (loop P B)).

B -- L -->> true
|-
(B -> P) -- L --> P.

G1 -- L --> G1'
|-
(G1 [] G2) -- L --> G1'.

G2 -- L --> G2'
|-
(G1 [] G2) -- L --> G2'.
```

Figure 25: SFC semantics

environment, *i.e.* open reactives programs must be closed. Notice that the user may use a *different* language to describe the environment. This is an interesting feature, since a more expressive language may be used to describe the environment.

5 Related Work

Our analysis algorithms are similar to the iterative data-flow analysis algorithms [Muc97] used in compiler construction. In iterative data-flow analysis, one represents a control-flow graph as a set of recursive equations. The fix point of the set of recursive equations produces the result of the analysis. However, such method may not be easily extended to interprocedural analysis or concurrent languages. Precision can be lost inadverted due to impossible paths when the analysis is based on the control-flow graph. All forward and backward analysis [Muc97] may be implemented in our framework as algorithms that traverse the abstract state space.

There are several frameworks based on denotational semantics [Nie82, VF92, DG81]. They base an intentional analysis on an extensional theory, impacting the specification of collecting semantics. The denotational semantic descriptions must be augmented by "Occurrence" or "Label" arguments, which label program points. Notice also that such frameworks are, in fact, based on an operational interpretation of the meta language (i.e. a variation of λ-calculus) used in the denotational specifications. The analysis process is implemented by converting the program to the respective λ-term (i.e. the program denotation), as this λ-term is augmented with the "occurrence" arguments, the "interpretation" of the λ-term produces the analysis result. Observe that λ-calculus could be compared with the intermediate representations used in our framework.

The Syntox [Bou90] system is an interprocedural abstract debugging system for a subset of Pascal. This system is implemented using several different techniques, namely: forward propagation, backward propagation, least fixed point computation and greatest fixed point computation. The program is represented by a recursive set of equations. Notice that when programs have recursive procedures, the interprocedural call graph is dynamically unfolded during the analysis, generating new equations. We believe that our approach based on model checking techniques is more precise for detecting errors in programs. The use of recursive equations extracted from the flow graph produces imprecisions which are not acceptable in verification procedures. It is also not clear how this method may be adapted to handle concurrent languages.

SPIN [Hol91, Hol97] is a state space exploration model checker. SPIN accepts design specifications written in the verification language Promela, and it accepts correctness claims specified in linear temporal logic. As mentioned before, several algorithms of our framework are based on ideas of SPIN. Promela is not an expressive language. For example, in [HLP97] several problems are described in order to map a system coded in a lisp-like language to Promela. For each specification, SPIN generates a C program that performs the desired verification. Therefore, the performance of the verifiers produced using SPIN are much better. However, we believe that after finishing the implementation of the PAN partial evaluator, we will reduce this difference. It should be mentioned that SPIN is intended to verify protocols, thus it is not adequate to detect errors such as dereferencing null pointers or invalid array accesses (i.e. index out of bounds), that may be easily handled by our tools.

VeriSoft [God97] is a state space exploration model checker for languages such as C or C++. Due to the complexity of the state, this system uses a state-less search algorithm. It is easy to see that cycles can not be detected in the context of a state-less search, but VeriSoft can always guarantee a complete coverage of the state space up to some depth. Once again, the system is only intended to verify protocols.

6 Conclusion

We have presented a framework for the development of program analysis and verification tools starting from modular formal definitions of language operational semantics. The theoretical foundations and algorithms have been described. We have also illustrated why the framework is particularly interesting for developing domain specific languages (DSLs). The framework comprises the logic programming language PAN, several state space exploration algorithms, a library of approximations, and reusable semantic descriptions (*i.e.* semantic "legos"). We also shown how a high-level and efficient implementation may be achieved. "Interpreters" produced through the framework may be used to perform simulations, verifiers are used to detect errors, and analyzers collect run-time information used to apply optimizing transformations to the code. Therefore, our framework can be used to increase the confidence in the correctness of a given program/specification, and to help the generation of optimized code starting from high-level specifications coded in DSLs.

Acknowledgments

The authors would like to thank Ira Baxter, Christiano Braga, and Michael Mehlich for their ideas, comments and contributions during the development of this work. We are also grateful to the National Research Council of Brazil (CNPq).

References

[ABBC99] Atkins, D. L., Ball, T., Bruns, G., and Cox, K. Mawl: A domain-specific language for form-based services. *IEEE Trans. Software Engrg.*, 25(3):334–346, 1999.

[Arn92] Arnold, A. *Finite Transition Systems.* Prentice-Hall, 1992.

[AT98] Anderson, S. and Tourlas, K. Design for proof: An approach to the design of domain-specific languages. *Formal Aspects of Computing*, 10(5&6):452–468, 1998.

[Bax92] Baxter, I. D. Design Maintenance Systems. *Comm. of ACM*, 35(4):73–89, April 1992.

[Bel94] Bell, J. Software Design for Reliability and Reuse: A proof-of-concept demonstration. In *Tri-Ada'94*, pages 396–404. ACM Press, November 1994.

[Bou90] Bourdoncle, F. Interprocedural abstract interpretation of block structured languages with nested procedures, aliasing and recursivity. In *PLILP'90*, volume 456 of *LNCS*, pages 307–323. Springer-Verlag, 1990.

[CC77] Cousot, P. and Cousot, R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *4th POPL, Los Angeles, CA*, pages 238–252, January 1977.

[CC92] Cousot, P. and Cousot, R. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *PLILP'92*, volume 631 of *LNCS*, pages 269–295. Springer-Verlag, 1992.

[CC93] Cordy, J. and Carmichael, I. The TXL programming language syntax and informal semantics. Technical report, Queen's University at Kinkston, Canada, 1993.

[CES86] Clarke, E. M., Emerson, E., and Sistla, A. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986.

[Cle88] Cleaveland, J. G. Building application generators. *IEEE Software*, July 1988.

[Cou96] Cousot, P. Abstract Interpretation. *ACM Comp. Surv.*, 28(2):324–328, June 1996.

[CRL99] Chandra, S., Richards, B., and Larus, J. R. Teapot: A domain-specific language for writing cache coherence protocols. *IEEE Trans. Software Engrg.*, 25(3):317–333, 1999.

[DG81] Donzeau-Gouge, V. Denotational definitions of properties of program computations. In Muchnick, S. S. and Jones, N. D., editors, *Program Flow Analysis: Theory and Applications*, chapter 11, pages 343–379. Prentice-Hall, 1981.

[DLNS98] Detlefs, D. L., Leino, K. R. M., Nelson, G., and Saxe, J. B. Extended Static Checking. SRC Research Report 159, Compaq Systems Research Center, December 1998.

[Fut71] Futamura, Y. Partial evaluation of computation process – An approach to a Compile-Compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[God96] Godefroid, P. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion*, volume 1032 of *LNCS*. Springer-Verlag, January 1996.

[God97] Godefroid, P. Model checking for programming languages using VeriSoft. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, January 1997.

[HLP97] Havelind, K., Lowry, M., and Penix, J. Formal analysis of a space craft controller using spin. Technical report, NASA, Ames Research Center, California, 1997.

[Hol91] Holzmann, G. J. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[Hol97] Holzmann, G. J. The model checker Spin. *IEEE Trans. Software Engrg.*, 23(5), May 1997.

[Hud96] Hudak, P. Building Domain-Specific Embedded Languages. *ACM Comp. Surv.*, 28(4), 1996.

[JGS93] Jones, N. D., Gomard, C. K., and Sestoft, P. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[JM81] Jones, N. D. and Muchnick, S. S. Flow analysis and optimization of LISP-like structures. In Muchnick, S. S. and Jones, N. D., editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, 1981.

[Lew95] Lewis, R. W. Programming industrial control systems using iec 1131-3. Technical report, Control Engineering. The Institution of Electrical Engineers (IEE), London, 1995.

[LSF94] Leite, J. C., Sant'anna, M., and Freita, F. G. Draco-PUC: a technology assembly for domain oriented software development. In *Proc. of the 3rd IEEE International Conference of Software Reuse*, 1994.

[Mil78] Milner, R. Theory of type polymorphism in programming. *J. of Computer and System Sciences*, 17(3):348–375, 1978.

[Mil89] Milner, R. *Communication and Concurrency*. Prentice-Hall, 1989.

[MLH00a] de Moura, L. M., de Lucena, C. J. P., and Haeusler, E. H. The PAN language. *submitted to ACM Transactions of Programming Languages and Systems*, 2000.

[MLH00b] de Moura, L. M., de Lucena, C. J. P., and Haeusler, E. H. Writing modular intepreters, analyzers and verifiers. *submitted to Software Practice & Experience*, 2000.

[Mos99] Mosses, P. D. Foundations of modular SOS (extended abstract). In *MFCS'99, Proc. 24th Intl. Symp. on Mathematical Foundations of Computer Science, Szklarska-Poreba, Poland*, LNCS, 1999. to appear.

[Muc97] Muchnick, S. S. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.

[Nie82] Nielson, F. A denotational framework for data flow analysis. *Acta Informatica*, 18:265–287, 1982.

[PE93] Plasmeijer, R. and van Eekelen, M. *Functional Programming and Parallel Graph Rewriting*. Addison Wesley, 1993.

[Plo81] Plotkin, G. D. A Structural approach to Operational Semantics. Technical Report FN-19, DAIMI, University of Aarhus, Denmark, September 1981.

[Pol54] Polya, G. *Mathematics And Plausible Reasoning*. Princeton Univeristy Press, 1954.

[RD92] Roy, P. L. V. and Despain, A. High-performance logic programming with the aquarius prolog compiler. *IEEE Computer*, 25(1):54–68, January 1992.

[Sah91] Sahlin, D. An automatic partial evaluator for full Prolog. Ph.D. thesis TRITA-TCS-9101, Kungliga Tekniska Hgskolan, Stockholm, Sweden, 1991.

[Sch98] Schmidt, D. A. Trace-based abstract interpretation of operational semantics. *J. Lisp and Symbolic Computation*, 10(3):237–271, 1998.

[SHC96] Somogyi, Z., Henderson, F., and Conway, T. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.

[Shi88] Shivers, O. Control flow analysis in Scheme. In *SIGPLAN '88 Conference on PLDI, Atlanta, Georgia*, volume 23(7) of *ACM SIGPLAN Not.*, pages 164–174, July 1988.

[SS94] Sterling, L. and Shapiro, E. *The Art of Prolog*. MIT Press, second edition, 1994.

[Ste91] Steffen, B. Data flow analysis as model checking. In *Proc. TACS'91*, volume 526 of *LNCS*, 1991.

[VF92] Venkatesh, G. A. and Fischer, C. N. Spare: A development environment for program analysis algorithms. *IEEE Trans. Software Engrg.*, 18(4), April 1992.

[Wad90] Wadler, P. Linear types can change the world! In *Proc. of the IFIP TC2 Conference on Programming Concepts and Methods*, pages 547–566, April 1990.

[Wei84] Weiser, M. Program slicing. *IEEE Trans. Software Engrg.*, SE-10:352–357, July 1984.