A Formal Analysis of Concurrent Assembly Code based on CSP

Gustavo Carvalho[1], Rafael Cabral[1], and Alexandre Mota[2]

[1]Programa de Engenharia de Computação - Universidade de Pernambuco
50720-001 Recife-PE, Brazil
{ghpc,rfc2}@ecomp.poli.br
[2]Centro de Informática - Universidade Federal de Pernambuco
50740-560 Recife-PE, Brazil
acm@cin.ufpe.br

Abstract. This paper presents guidelines for using the CSP process algebra, particularly the CSP# variant, to describe the behaviour of concurrent Assembly code. To properly model the Assembly code behaviour, aspects of the runtime environment are formally described as well. This work considers the set of instructions of the x86 Assembly language, and the Windows environment for threads support. We also demonstrate how the created CSP# specification and Linear Temporal Logic (LTL) formulae can be used for analysing concurrent properties of two classical problems: the Producer/Consumer and the Dinning Philosophers. The preliminary results show that this strategy might be used for the formal analysis of concurrent Assembly code in general.

Keywords: CSP, Process Algebra, Assembly, Threads, x86

1 Introduction

In the last years, we had a significant increase of embedded HW-SW components. This trend increases software size and complexity, and impacts their safety and reliability. Testing is one of the approaches for developing systems with higher quality levels. However, it cannot guarantee the absence of errors, particularly in concurrent settings. For example, race conditions are difficult to detect since testing does not usually take into account the processor scheduling decisions. Besides that, there is a gap between the high-level language that is analysed and the low-level language that really provides the system's behaviour. To detect these problems we can use formal techniques based on the machine code level.

In this work we use the CSP process algebra (indeed its CSP# variant [4]) to describe the behaviour of concurrent Assembly code. We consider the x86 platform and the Windows Memory/Thread API. However, our proposal can be used in other CISC architectures as well. Our main contributions are: (1) guidelines for representing the x86 runtime environment, the x86 Assembly language, and part of the Windows Memory/Thread API in CSP#, (2) how LTL can be used to express deadlock and livelock properties concerning the machine code of the classical Producer/Consumer and Dinning Philosophers implementations.

The next section presents the language CSP#. Section 3 describes how the x86 runtime environment can be represented in CSP#. Section 4 details how concurrent x86 Assembly code can be represented in CSP#. Section 5 shows how the created CSP# code can be used for analysing properties. Section 6 addresses related work, and presents our conclusions and future work.

2 CSP#

CSP is a formal language for describing behavioural aspects of systems. Its fundamental element is the concept of process. For practical purposes, CSP has some machine readable versions. Here, we consider the CSP# variant because it naturally supports communication via shared memory, and its tool, the Process Analysis Toolkit (PAT), has the ability to perform LTL on-the-fly model checking. A process behaviour is described by the set of events and the ordering it can communicate these events to other processes. An event is an atomic transition that changes the process state. A linear behaviour is obtained with the prefixed process ($ev \rightarrow P$), where ev is an event and P a process. CSP has two primitive processes: $Skip$ represents successful termination and $Stop$ captures abnormal termination, also interpreted as a deadlock.

In CSP#, we define constants with the $\#define$ operator. To declare global shared memory variables we use the syntax ($var\ v = initial\_value;$). To change part of the memory, we use a data operation event ($ev\{v = new\_value; ...\}$, where ev is an event). All assignments inside a data operation are performed sequentially and atomically. CSP# also supports arrays/matrices.

To define alternating behaviours, we use the conditional choice operator (if) as it is similar to the ones found in standard programming languages. To close our CSP# introduction, we still have the sequential composition $P() = P1(); P2();$ stating that P behaves as $P1()$ and consequently as $P2$, when $P1()$ terminates successfully. Concerning parallel composition, CSP# allows a composition with ($P1||P2$) or without ($P1|||P2$) synchronisation. To make the behaviour of a whole process atomic, we use the term $atomic\{P()\}$.

3 The x86 Runtime Environment in CSP#

The x86 runtime environment is composed of, for each program, from lower memory addresses to higher memory addresses, the following logical constituents in this specific order: the program binaries, static data, the program heap and the program stack. The stack is further decomposed into frames, which represent different program scopes. The register ebp stores the base address of the last frame, whereas the register esp has the address of the last frame top. In this work we assume a 32bit architecture. When the program has more than one thread, each thread has its own stack, but the heap is shared among all threads. We suggest reading [1] for more details about this runtime environment.

In our CSP# model, the static data and global variables are both represented as CSP# global variables. Program binaries are encoded as CSP# processes. In what follows consider n ($n > 0$) the number of threads. The individual stacks and the shared heap are both represented by a single matrix (*memory*) of size $(n{+}1){\times}m$, where m ($m > 0$) is set to the maximum size of each memory fragment (stack or heap). The first n dimensions of this matrix keep the threads' stacks, whereas the last one stores the heap (Figure 1.1, lines 1–2).

For each register (eax, ebx, eip, etc.) we have a CSP# array of size n. Almost all registers are initialized with zero. The exceptions are ebp and esp that initially points to the stack base (variable MEM_LAST_INDEX is set to $m - 1$; not shown). In the line 2 we use the notation $[MEM\_LAST\_INDEX(n)]$ for creating an array of size n initializing each position with MEM_LAST_INDEX.

Figure 1.1. CSP# - Program Stacks and Heap

```
1 #define m 30; #define n 3; var memory[n + 1][m];
2 var eax[n]; var ebp = [MEM_LAST_INDEX(n)]; var esp = [MEM_LAST_INDEX(n)];
```

4 The x86 Assembly Instructions in CSP#

Before presenting how x86 Assembly instructions are encoded in CSP#, we need to understand how some operations of the Windows Memory/Thread API are modeled in CSP#. Let's first consider the heap memory (de)allocation. This is done by calls to the *_free*/*_malloc* functions of the Windows API. For *_malloc*, the runtime environment stores the current heap top address in eax, considering the running thread context, and increments the heap top by the space requested. Figure 1.2 shows the CSP# process (*Malloc*) that captures this behaviour. The variable *current_heap* represents the heap top, and *id* is the identifier of the current thread. It is worth noting that the heap top is incremented by 4 bytes (32bits) because we are dealing only with 32bit variables. The deallocation (process *Free*, omitted here to save space) is defined analogously.

Figure 1.2. CSP# - Heap Memory Allocation

```
Malloc(id, size) = call_malloc.id { eax[id] = current_heap;
                         current_heap = current_heap + size/4; } -> Skip;
```

A thread is created by an external call to the function *_CreateThread* of the Windows API. Although this function has six parameters, we consider just one that refers to a label in the Assembly code from which the new thread will start executing. Our CSP# translation uses the same name (Figure 1.3) but two parameters. The first one represents the previously mentioned label, and the second one is the identifier of the thread that called *_CreateThread*. This keeps in the caller thread eax the result of the *_CreateThread* invocation. To simulate threads wait and resume operations, we use the array (*thrdState*) to store each thread current state (0 - suspended, 1 - running, and 2 - terminated). The main thread (the one related to the program entry) is always initiated with 1.

As race conditions occur during thread execution, and not during thread creation, we do not consider scheduling during thread creation, making the CSP# process for creating threads atomic with the *atomic* operator (line 2). The event *call_CreateThrd* increments the number of created threads, and it properly updates the registers (omitted here). Then, the state of the new thread is changed to running (line 3). After that, it performs the event *start.proc.current_id* that triggers the start of the current thread by means of a parallel synchronization. We also have modelled thread suspension/resuming/termination, as well as mutexes. However, due to space restrictions, we omitted them here.

Figure 1.3. CSP# - Thread Creation

```
1 var thrdState = [1, 0(MAX_THREADS)]; var cur_id = 0;
2 _CreateThread (proc, creatorId) = atomic{ call_CreateThrd{cur_id++; ...}
3    -> setThrdState.id{thrdState[id]=1} -> start.proc.current_id -> Skip};
```

We also need a specific structure for modeling the parallel execution of the Assembly instructions. Let $L1$ and $L2$ be two labels corresponding to instructions that must be run in parallel. The code we produce is shown in Figure 1.4. Each label creates a process. The behaviour of them conforms to the sequence of instructions associated with each label. Each process has one parameter ($id > 0$) that represents the id of the thread invoking the respective label.

Figure 1.4. CSP# - Parallel Behaviour

```
\\ ——————————————————————— x86 Assembly ———————————————————————
L1: ... (instructions A)
    _CreateThread
    ... (instructions B)
L2: ... (instructions C)
\\ —————————————————————————— CSP# ——————————————————————————
L1(id) = L1_Body(id) || L2(1);
L1_Body(id) = ... \\ the CSP# equivalent of instructions A
              -> _CreateThread(L2__Proc, 0) ;
              ... \\ the CSP# equivalent of instructions B
L2(id) = start.L2__Proc.id -> L2_Body(id) ; _ExitThread(id);
L2_Body(id) = ... \\ the CSP# equivalent of instructions C
```

Finally, Table 1 shows how the most common x86 Assembly instructions are translated into CSP#. The translation of arithmetic instructions is quite straightforward (Table 1, lines 1–2). The only exception is when the second operand is ebp or esp. In such cases (line 3), the first operand (usually a number) is divided by 4; recall that we are dealing with 32bit values (4 bytes). The *cmp* instruction corresponds to atomically subtracting the operands, and storing the result in the register $flags$. In CSP# we model the register $flags$ via the array *cmps* whose size equals to the number of threads (Table 1, line 4). The conditional jump consists of a logical test on the register $flags$. In CSP# this is done analogously (Table 1, line 5). Non-conditional jumps (*jmpLabel*) are just translated to the corresponding CSP# process for the label (*Label*(*id*)). The *mov* instruction moves data from the first operand to the second one. For instance, Table 1, line 6, shows the assignment of the value pointed by eax to the ebx register. An Assembly code also has stack operations like *push*, *pop*,

Table 1. Translation of Assembly Instructions to CSP# Code

Line	Assembly	CSP#
1	add $1, %eax	_add { eax[id] = eax[id] + 1 }
2	imul %ebx %eax	_imul { eax[id] = ebx[id] * eax[id] }
3	add $4, %esp	_add { esp[id] = esp[id] + 1 }
4	cmp %eax, %ebx	_cmp { cmps[id] = eax[id] - ebx[id] }

5	jle Label
	... (A)
		if (cmps[id] <= 0) { Label(id) }
		else { ... (CSP# equivalent to A) }

6	mov (%eax), %ebx	Label_mov { eax[id] = memory[id][eax[id]] }
7	push %ebp	_push.id { esp[id]--; memory[id][esp[id]] = ebp[id] }
8	leave	_leave.id { esp[id] = ebp[id]; ebp[id] = memory[id][esp[id]]; esp[id]++ }
9	ret	Skip

enter, and *leave*. The instruction *enter* (not shown) performs a *push* and a *mov* atomically to create a new stack frame. The instruction *leave* does the opposite of the *enter* instruction. Table 1, lines 7–8, shows the CSP# for *push* and *leave*. Finally, the instruction *ret* is mapped to the CSP# *Skip* process (line 9).

5 Analysing Properties

From a CSP# specification, the PAT tool can check properties, such as: (1) deadlock-freedom, (2) deterministic behaviour, (3) divergence-freedom, (4) successful termination, and (5) if a process satisfies some LTL formulae. To preliminary evaluate our strategy, we created a CSP# specification considering the x86 Assembly generated by a C compiler for standard implementations of the Producer/Consumer and the Dinning Philosophers classical problems.

For the Producer/Consumer problem, the PAT tool reports it is not deadlock-free. By examining the trace given by PAT, we can identify the events corresponding to its well known deadlock: the consumer decides to sleep, but before sleeping the producer tries to wake up the consumer. The wake up is lost, the consumer sleeps, the producer produces until the buffer limit, and sleeps as well.

An alternative assertion is to check the following LTL formula: *#define goal (thrdState[1] == 0* && *thrdState[2] == 0); #assert _main() |= !(<> goal)*, where *_main* is the process that represents the label for the entry-point of the Assembly code, and *goal* a situation where both threads (producer id = 1, consumer id = 2) are in the suspended state (0). The LTL formula states that this situation never happens in the future. As it is not true, PAT returns a counter-example (trace) equals to the one returned by the deadlock-free assertion.

Concerning the Dinning Philosophers problem, it is expected that all philosophers will eventually eat in the future. This property can be expressed by the following LTL formula: *#assert _main() |= (<> call_eat.1)* && *(<> call_eat.2)* && *(<> call_eat.3)*. The events *call_eat.i* ($i \in \{1, 2, 3\}$) represent the jump to the label *_eat*, which is associated with the eating code for each thread (*.i*). PAT also returns a counter-example for this formula that depicts the situation where each philosopher tries to pick one of its forks and thus they all block forever.

PAT took between 0.12s and 2.14s to check the assertions for the first and second examples. Therefore, for these examples, our CSP# representation of the concurrent Assembly code was adequate to quickly discover concurrent problems that are not easily detected by traditional testing techniques.

6 Conclusions

This paper reports an ongoing work that describes in CSP# the behaviour of concurrent Assembly code. We are considering the x86 runtime environment, and the Windows API. For classical examples of concurrent problems (Producer/-Consumer and Dinning Philosophers), the CSP# representation of x86 Assembly allowed the discovery of the well-known problems within seconds.

The idea of formal analysing Assembly code has already been explored by similar works. The Vx86 tool [3] is a static analyser that verifies the correctness of x86 Assembly by simulating it in C with the help of the Z3 SMT solver. This work verifies properties not currently supported by our strategy, like absence of overflows. However, the Vx86 seems to be able to verify only sequential code.

Based on the LLVM (Low Level Virtual Machine), the work reported in [2] describes the LLVM2CSP tool that extracts CSP models for the ProBE and FDR tools. The main difference to our work is that the LLVM2CSP requires the C/C++ to be annotated with particular calls. This is not required by our approach. However, concerning the LLVM2CSP tool, the user can define by means of annotations domain specific abstractions to deal with the state explosion problem. This cannot be done in our approach yet. Another similar work is the one presented in [5]. It describes the Vellvm (Verified LLVM) framework for reasoning about programs described in the LLVM's intermediate representation. This framework is built using the Coq interactive theorem prover. Thus, it addresses the verification of concurrent code by means of theorem proving, whereas our approach focus on model checking.

Some practical aspects are still not supported yet, like floats and user input data, and our empirical evaluation is preliminary. Therefore, we foresee the following future works: enhance/formalize our translation, investigate the soundness of our strategy, study compositional analysis, refinement of concurrent code, and apply the strategy for bigger and real examples.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Prentice Hall (2006)
2. Kleine, M., Bartels, B., Gthel, T., Helke, S., Prenzel, D.: Extracting CSP Models from Concurrent Programs. In: NFM. LNCS, vol. 6617, pp. 500–505 (2011)
3. Maus, S., Moskal, M., Schulte, W.: Vx86: x86 assembler simulated in c powered by automated theorem proving. In: Proceedings of AMAST (2008)
4. Sun, J., Liu, Y., Dong, J.S.: Model checking CSP revisited: Introducing a process analysis toolkit. In: ISoLA Proceedings. pp. 307–322. Springer (2008)
5. Zhao, J., Nagarakatte, S., Martin, M.M., Zdancewic, S.: Formalizing the LLVM intermediate representation for verified program transformations. SIGPLAN Not. 47(1), 427–440 (2012)