thr2csp: Toward Transforming Threads into Communicating Sequential Processes

Robert Lange and Spiros Mancoridis
Software Engineering Research Group
Department of Computer Science
College of Engineering
Drexel University
3141 Chestnut St., Philadelphia, PA 19104
{rcl24,spiros}@drexel.edu

***Abstract*—As multicore and heterogeneous multiprocessor platforms replace uniprocessor systems, software programs must be designed with a greater emphasis on concurrency. Threading has become the dominant paradigm of concurrent computation in the most popular programming languages. Large threaded programs are known to be difficult to implement correctly, comprehend, and maintain, while concurrent programs written in process algebraic paradigms of concurrency, such as communicating sequential processes, are known to be easier to analyze. This paper presents our initial work on reverse engineering threaded source code and transforming the code into functionally-equivalent message-passing code. The paper also explores future work needed to convert the message-passing code into communicating sequential processes.**

I. Introduction

As multicore and heterogeneous multiprocessor platforms become ubiquitous, more of the execution of software programs must occur concurrently in order to take advantage of the new platforms. While concurrency is used to improve software execution speed, maximum execution speed is not always the primary requirement of a software system. Concerns such as correctness, efficient power consumption and heat distribution, and responsiveness (*i.e.*, timing predictability) may be more important requirements for a particular software system than speed. The implementation of concurrent software in a form that enables it to meet these requirements is a growing concern for software engineers.

The *multithreading* paradigm of concurrency [1], [2] has become the dominant paradigm used in the most popular languages, such as C, C++, and Java. However, threads are known to be difficult to write and analyze [3]. Process algebraic paradigms, such as communicating sequential processes (CSP) [4], are known to be more amenable to program analysis. However, process algebras languish in attention from language designers, system programmers, and application programmers due largely to the dominance of multithreading. Most concurrent programs today are written using threads and most languages and development libraries include threading as the default concurrency mechanism; therefore, students are more likely to demand to learn multithreading, schools are more likely to teach multithreading, and the cycle repeats.

This paper explores the possibility of using source-to-source program transformation to transform threaded programs into CSP programs. The initial work transforms threaded programs into message-passing programs. Additional transformational steps that render the program more CSP-like are described. Finally, we identify how user involvement is necessary to complete the transformation of a multithreaded program into a proper CSP program.

II. The Problems with Threads

Multithreading is a shared-memory paradigm of concurrency in which communication between threads occurs by writes to shared memory regions. These writes are indistinguishable, via local program analysis, from non-communicating writes. After every atomic instruction is executed, the executing thread may be interrupted and another thread may begin or resume execution. Explicit concurrency control is necessary to guard *critical sections* of code, *i.e.*, sections of code that must be executed without interfering communications from other threads. As a consequence of these properties, threads are not composable [3]. When two or more threads are combined in one program, whole program analysis is necessary to determine the interactions between the threads. In practice, when new threads are added to a program, changes are often required to the existing threads.

Threads are known to be difficult to analyze [3]. Whole program analysis is necessary to determine which threads communicate [5] and which portions of the code may execute in parallel [6], [7]. These analyses are relatively expensive (worst case $O(n^3)$ time) in the size of the program. Dependence analysis and program slicing of shared memory concurrent programs is undecidable in general [8]. Conservative approximations require special consideration for threaded programs to accommodate inter-thread communication [9], [10], [11]. Krinke and Nanda's relatively-precise dependence analyses execute in exponential time in the worst case, although Giffhorn provides a more efficient, but less precise, context-insensitive variant of Nanda's dependence analysis [12].

The expense of static analysis of threaded programs has resulted in the prevalence of manual inspection and *ad hoc* testing for the demonstration of correctness of concurrency. Even with a rigorous test suite, faults due to the occurrence of unexpected thread interleavings go unnoticed. Lee speculates that as multicore and heterogeneous multiprocessor architectures become more common, formerly correctly-behaving software will fail due to the greater opportunities for truly concurrent execution, and thus the greater opportunities for exploration of unexpected interleavings [3].

A. Communicating Sequential Processes

As the name implies, communicating sequential processes programs are systems of processes, in which the logic of each process is sequential, and in which all inter-process communication occurs via explicit communication events. Hoare introduced the concept informally [4], and it was subsequently formally analyzed [13], [14], [15]. For the purposes of discussion in this paper, we use Schneider's definitions and notation [16].

In CSP, one process cannot perform an action that, as a side-effect, affects another process's exection. This property enables the composition of processes into higher-level processes, similar to functional programs. Processes can be nested within other processes, and the inner workings of a process are not visible or relevant to other processes. These properties of CSP programs are in constrast to multithreaded programs, which are not composable without significant programmer discipline. Furthermore, the flow of data through a CSP program is a property of the language, whereas with multithreading, it is a side-effect, the understanding of which is expensive to determine.

Techniques exist to support the refinement of CSP specifications and to prove whether properties of the specification hold in the refined program [15], [17], including deadlock and livelock-freedom.

III. Transforming Thread Operations

The ultimate goal of the *thr2csp* project is to transform a threaded program into a CSP program, which communicates only through message-passing. This is a multi-stage process, and this paper details the initial stage of the process. In the initial stage, thread creation and communication constructs are transformed into message-passing constructs with equivalent semantics.

A. C++CSP

The *thr2csp* project has as its initial source language POSIX threads in the C and C++ programming languages. Therefore, the transformed code uses the *C++CSP* library [18], [19] as the target for transformation. C++CSP implements a CSP-based framework on top of the C++ language. C++CSP provides *processes* and *channels* as the two primitives, which are described in this section.

C++CSP offers channels as an abstraction for the communications event types in CSP. Channels in C++CSP are typed, bi-directional communications routes between processes, and may be one-to-one, one-to-any, any-to-one, or any-to-any. The one-to-any, any-to-one, and any-to-any channels implement interleaving semantics of synchronization ($|||$) such that, from the set of processes that may observe a specific communication, only one process actually synchronizes and observes the event. C++CSP processes communicate over channels via "channel ends"; a process reads from input channel ends and writes to output channel ends.

In CSP, communication is synchronous, requiring both sender and recipient to block until the other is ready to engage in the communication. Asynchronous communication may be implemented by means of a FIFO buffer process acting as an intermediary to the communicating processes; with this implementation, a sender may write data, as long as the buffer is not full, and subsequently continue execution even when the recipient is not ready to read the data. C++CSP channels may be *synchronous* or *buffered*. A buffered channel may be finite, in which case the writer will block if the buffer is full, or *overwriting*, in which case, when the buffer is full, the oldest member of the buffer is overwritten.

The channel alternative mechanism, known as *alting*, provides the notion of nondeterministic, or internal, choice ($\sqcap$) to processes. A process may alternate over several channels, and from the set of ready channels, one will be chosen for synchronization based on selection criteria. Criteria for the choice that are available in C++CSP include priority, round-robin, and random selection.

C++CSP offers several mechanisms for barrier control, all deriving from the CSP parallel composition ($||$) construct, in which all processes involved in a communication event must be ready in order for the communication to occur. Of relevance to this research is the *bucket synchronization* method. In terms of CSP, a bucket is a communication event that is a parallel composition of a writer process with the union of a set of reader processes and the *READY* process, which is a process that is always ready to communicate. All processes which "fall into" the bucket block until the bucket is "flushed" by a writer process. This mechanism is useful for modeling condition variables.

Finally, C++CSP provides several options for controlling the sequence and parallelism of processes. The method used by *thr2csp* to fork processes is the *ScopedForking* object. ScopedForking objects allow the C++CSP programmer to invoke sets of parallel processes. When the instance of the ScopedForking object falls out of scope, the parent process blocks until all processes forked by the instance terminate.

B. Thread Operations to Channel Operations

The initial stage of the transformation of a threaded program into a CSP program implements a one-to-one conversion of threading communication constructs into message-passing constructs. Three primitive POSIX thread communications operations have been identified for transformation: reads from and writes to shared memory regions, mutex handling, and wait-signal semaphores. These operations can be performed in a message-passing environment using specialized communications channels built on top of the built-in C++CSP channels.

1) Shared Memory Channels: In a threaded program, the primary method of communication between threads is by shared regions of memory, *i.e.*, pointers to variables or arrays. When a thread executes an atomic write operation to place value a in a memory location X, any subsequent read operation by any thread on X will observe value a, until the next write operation on X.

The semantics of an atomic read/write can be simulated by an **SHMChannel** process, which consists of the composition of an any-to-one input channel and a one-to-any output channel. For each shared variable X, an SHMChannel S_X is instantiated. Each write to X is substituted by a write to the input channel of S_X and each read from X is substituted by a read from the output channel of S_X.

Properties of a write to shared memory are that it is asynchronous, *i.e.*, once the write has occurred, the thread may continue executing, regardless of whether another thread has read the value; and that it is persistent, *i.e.*, the value written to the memory will remain in memory until another write replaces it. These properties are simulated in the SHMChannel by the composition of two processes, the

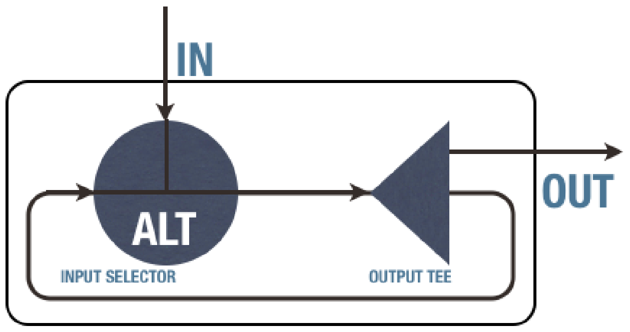

Figure 1. A visual depiction of the SHMChannel process. This channel implements an asynchronous overwriting buffer with persistence. The ALT prefers input from the IN channel when available, ensuring that the newest value is always propagated to the OUT channel.

InputSelector and the **OutputTee**. The InputSelector has two inputs and one output; the OutputTee has one input and two outputs. One input of the InputSelector is connected to the input to the SHMChannel, while the other is connected to an output of the OutputTee. The output of the InputSelector is connected to the input of the OutputTee, and the other output of the OutputTee is connected to the output of the SHMChannel. The SHMChannel is depicted in Figure 1.

The InputSelector blocks until input is received from either the SHMChannel or from the OutputTee; if input is received from both, the input from the OutputTee is discarded, and the input from the SHMChannel is chosen and passed to the OutputTee. When a value is read from the OutputTee by the SHMChannel, that value is also written back to the input of the InputSelector. This arrangement guarantees both the properties of asynchrony and persistence for the SHMChannel.

2) Mutex Channels: In a shared-memory concurrency environment, some form of locking is necessary to ensure that non-atomic operations can occur without interference. In POSIX threads, the primitive operation given to implement this is the mutual exclusion lock, or mutex. Semantically, many threads may hold a reference to a mutex, but only one thread can hold the lock to the mutex, and once locked, only that thread can unlock the mutex.

Mutexes are simulated with the **LockChannel** process, which consists of the composition of one synchronous any-to-one input channel and one synchronous one-to-any output channel. Reading a token from the LockChannel process's output corresponds to locking a mutex, while writing the token back to the input channel corresponds to unlocking the mutex.

Internally, the LockChannel consists of a composition of two processes, a **LockListener** and an **UnlockListener**. The input channel of the LockChannel is connected to the input channel of the UnlockListener, and the output channel of the LockListener is connected to the output channel of the LockChannel. The output of the UnlockListener is connected to the input of the LockListener.

The LockListener blocks waiting for a token from the UnlockListener; upon arrival, it offers to communicate that token to the first process that requests it, then loops back to waiting for communication from the UnlockListener. Upon initialization, the UnlockListener initializes the token to a random non-zero value, then transmits that token to the LockListener. The UnlockListener then waits for an external process to communicate the token back to it. When the correct token arrives, the UnlockListener chooses a new non-zero token and transmits it to the LockListener; if an incorrect token arrives, the UnlockListener discards it.

`pthread_mutex_trylock` can be simulated with a minor change: if the lock cannot be acquired, the value returned is 0; for a successful lock, the return value is the non-zero token value.

3) Signal Channels: The semaphore mechanism offers a method for causing one thread to wait for a signal from another thread. In POSIX threading, when a thread waits on a condition variable (*i.e.*, semaphore), it specifies the condition variable on which it is waiting, and a locked mutex; the mutex is unlocked when the thread begins waiting. When another thread sends a signal over the condition variable, all of the threads waiting on the condition variable contend for the signal, and at least one waiting thread is woken to receive the signal.[1] Each woken thread then blocks until the mutex lock that it had unlocked when it began waiting is re-acquired.

Therefore, the POSIX procedure call `pthread_cond_wait` can be normalized to three logical procedures: unlock the mutex; wait on the semaphore; lock the mutex. This operation maps to a write to the LockChannel corresponding to the mutex to be released, a read from the bucket corresponding to the condition variable, and a read from the LockChannel. Likewise, `pthread_cond_signal` and `pthread_cond_broadcast` correspond to a write to the bucket.

[1]Most users of POSIX threads assume that `pthread_cond_signal` wakes exactly one sleeping thread, and this may be true for some implementations. However, the standard specifies "at least", therefore an implementation can wake more than one sleeping thread. By contrast, `pthread_cond_broadcast` is required to awaken all sleeping threads.

C. Transformation of Thread Operations

POSIX threads are not signified in C by any special keyword, nor are they implemented as a special type. Almost any procedure in a program may be the entry point for a new thread. POSIX threads are created using the `pthread_create` procedure, so all information about threads, including their entry points and their shared memory regions, must be extracted based on these procedure calls.

1) Creation of Channels: In a call to `pthread_create`, the fourth parameter, `arg`, specifies a pointer to information to be passed to the newly-created thread. This is typically a pointer to a `struct` containing shared variables or pointers to other shared memory regions. In this work, all shared memory is assumed to be passed into a thread explicitly via this mechanism. Pointer and escape analysis can be used in future work to eliminate this assumption.

The channel creation algorithm enters the `struct` pointed to by `arg` and creates, for each primitive variable X with type T, an SHMChannel S_X capable of transmitting messages of type T. For each mutex M a LockChannel L_M is created, and for each condition variable C, a bucket B_C is created.

Non-primitive types, such as structs, are handled recursively; this produces a fine-grained model of communication, albeit a messy model. Currently, pointer types are handled naïvely, which could result in out-of-channel communication between processes. A more detailed transformation would require pointer analysis, which is beyond the scope of this initial work.

2) Creation of Processes: The third parameter of `pthread_create`, `start_routine`, identifies the procedure call to be used as the thread entry point. The signature of procedures that can be thread entry points is `void* foo(void*)`, although casting can permit a variety of other signatures. Using the `pthread_create` procedure and the list of channels created in Section III-C1, enough information exists to create a C++CSP process.

A C++CSP process primarily consists of a set of channel ends and a `run` method. First the process creation algorithm lays down a boilerplate C++CSP process class definition, with the thread start routine's name as the name of the class. The thread start routine's body is transformed into the body of the class's `run` method. This method is analyzed to identify the uses of shared memory. For each shared variable X that is read from, the output channel end of S_X is included in the class's set of channel ends. Likewise, for each shared variable X that is written to, the input channel end of S_X is included. For each mutex M used in the body, both channel ends of L_M are included in the class. For each condition variable C, the bucket B_C

is included.

3) Transformation of the Process Body: To complete the transformation from thread to process, reads from and writes to shared memory must be replaced with reads from and writes to channel ends. The process class `run` method is analyzed to identify the uses of shared memory. For each shared variable X, a local variable lcl_X is created within the method.[2]

At each statement in which a shared variable X is read, a statement is prepended to read from the input channel end S_X into the local variable lcl_X, and the read of X within the statement is replaced with a read of lcl_X. Likewise, for each statement that writes to a shared variable X, the write to X is replaced with a write to lcl_X and a statement that writes lcl_X to the output channel end of S_X is appended.

For each mutex M, a local variable lcl_M is created. Each lock of mutex M is replaced by a read on the input channel end of L_M into the local variable lcl_M. This has the effect of causing the code to block until the lock is available. lcl_M contains the current lock token. Each unlock of the mutex M is replaced by a write of the variable lcl_M to the input channel end of L_M.

Each signal to C is replaced by a call to flush the bucket B_C. Each wait to C with the corresponding mutex M is replaced by an unlock over the channel L_M, a fall into the bucket B_C, and a lock over the channel L_M.

4) Forking Processes: The final stage of the process of adapting a threaded program into a message-passing program is to transform the invocation of the thread into the invocation of the process.

POSIX threads are handled by an object of type `pthread_t`. Each variable t of type `pthread_t` is converted to an instance of a C++CSP `ScopedForking` object P_t declared on the heap. A call to `pthread_create` for thread t is replaced by a call to the `fork` method of the `ScopedForking` object P_t.

When a `ScopedForking` object falls out of scope, its parent process blocks until all of the processes that it forked terminate. In the case of a heap allocation, this occurs when the `ScopedForking` object is deleted. Therefore, a call to the POSIX thread procedure `pthread_join` on thread t is equivalent to deleting the `ScopedForking` object P_t.

D. Example

The transformation methods described herein have been implemented using the Stratego source-to-source transformation language [20] with a custom C/C++ parser

[2]The reliance on local variables is due to the procedure-style pass-by-reference calls of C++CSP methods, and we intend to replace these with function-style read and write calls in the future, eliminating the need to declare local variables.

```
#include <stddef.h>
#include <stdio.h>
#include <pthread.h>

typedef struct shared_
{
  int x;
  int xst;
  pthread_mutex_t xm;
  pthread_cond_t xcv;
} shared_t;

void* thr1(void* arg)
{
  shared_t* a = (shared_t*) arg;
  pthread_mutex_lock(&a->xm);
  a->x = a->x + 100;
  a->xst = 1;
  pthread_cond_signal(&a->xcv);
  pthread_mutex_unlock(&a->xm);
  pthread_exit(NULL);
}

int main(int argc, char** argv)
{
  shared_t s;
  pthread_t thr_1;
  pthread_mutex_init(&s.xm, NULL);
  pthread_cond_init(&s.xcv, NULL);
  s.x = 0;
  s.xst = 0;
  pthread_create(&thr_1, NULL, thr1,
                 (void*) &s);
  pthread_mutex_lock(&s.xm);
  if (s.xst != 1) {
    pthread_cond_wait(&s.xcv, &s.xm);
  }
  pthread_mutex_unlock(&s.xm);
  pthread_join(thr_1, NULL);
  printf("%d\n", s.x);
  pthread_mutex_destroy(&s.xm);
  pthread_cond_destroy(&s.xcv);
  return 0;
}
```

Figure 2. A simple threaded example. The main thread sets a value, the thread modifies the value, and the main thread subsequently reads the modified value. Note that the `struct shared_t` acts as a vehicle to insert several variables into the thread start routine, and does not necessarily imply a logical relationship between the variables so passed.

and disambiguator. Auxiliary program analysis has been performed using the CodeSurfer program analysis tool for C and C++ [21], [22].

The example multithreaded code in Figure 2 shows a simple flow of information from a main thread, to a subordinate thread, and back again. The data relevant to the program is transmitted via the `x` variable. A mutex `xm` and a condition variable `xcv` are used to control the interleavings of the two threads in the program. The variable `xst` is not relevant to the high-level logic of the program, but acts to regulate the sequence of concurrency operations; because POSIX threading offers no operators to define the structure or interactions between threads, low-level mechanisms such as this are common. One of the main challenges of multithreading analysis is to separate the variables that contain program-relevant information from those that are used exclusively for concurrency control.

Figures 3 and 4 show the result of the application of the base *thr2csp* transformation. The `thr1` class shown in Figure 3 is the process corresponding to its namesake thread start routine in Figure 2. The process's constructor initializes its channels. In the `run` method, the points of possible communication are clearly defined by the use of the channels.

IV. Toward CSP

The C++CSP program in Figures 3 and 4 is not significantly easier to understand or maintain than the original multithreaded program in Figure 2. This is due to the fact that the initial *thr2csp* transformation is merely a one-to-one mapping of multithreading constructs to semantically-equivalent CSP constructs. The real power of CSP as a mechanism for concurrency is in its elegance at specifying the structure and interactions between concurrent processes, properties which a one-to-one mapping alone cannot identify.

For a program like that specified in Figure 2, an ideal CSP specification might be something like the following:

$$PROG = MAIN_1 ||| THR1 ||| MAIN_2$$
$$MAIN_1 = x_1!0 \rightarrow SKIP$$
$$THR1 = x_1?l \rightarrow x_2!(l + 100) \rightarrow SKIP$$
$$MAIN_2 = x_2?l \rightarrow stdout!l \rightarrow SKIP$$

In this CSP specification, $MAIN_1$ writes to channel x_1 and $MAIN_2$ reads from x_2, as opposed to a single process $MAIN$ writing to then reading from the same channel x, which could have ambiguous interpretations.

The CSP specification makes the structure and the interactions between the threads plain. The rest of this section will focus on methods to convert naïve transformations, like that in Figures 3 and 4 into something more resembling the ideal CSP specification.

```
 1 class thr1 : public  csp::CSProcess
 2 {
 3 private:
 4   csp::Chanin<int> x_in;
 5   csp::Chanout<int> x_out;
 6   csp::Chanout<int> xst_out;
 7   csp::Chanin<int> xm_in;
 8   csp::Chanout<int> xm_out;
 9   csp::Bucket xcv;
10 protected:
11   void run()
12   {
13     int lcl_x;
14     int lcl_xst;
15     int lcl_xm;
16     xm_in.read(&lcl_xm);
17     x_in.read(&lcl_x);
18     lcl_x = lcl_x + 100;
19     x_out.write(&lcl_x);
20     lcl_xst = 1;
21     xst_out.write(&lcl_xst);
22     xcv.flush();
23     xm_out.write(&lcl_xm);
24   }
25 public:
26   thr1(const csp::Chanin<int>& x_in_,
27        const csp::Chanout<int>& x_out_,
28        const csp::Chanout<int>& xst_out_,
29        const csp::Chanin<int>& xm_in_,
30        const csp::Chanout<int>& xm_out_,
31        const csp::Bucket& xcv_)
32   {
33     x_in = x_in_;
34     x_out = x_out_;
35     xst_out = xst_out_;
36     xm_in = xm_in_;
37     xm_out = xm_out_;
38     xcv = xcv_;
39   }
40 };
```

Figure 3. The output of the basic transformation of the thread function to a C++CSP process. Lines 13-23 make up the body of the process, while the remaining lines are boilerplate declaration and initialization of the channel ends.

A. Channel Splitting

CSP programs can be depicted using network graphs, as shown in Figure 5. In a traditional CSP network graph, processes are represented as nodes and communications events are represented as arcs; because our SHMChannels are actually special processes masquerading as channels, we represent them as special square nodes.

Ideally, the structure of the network graph should provide insight into the nature of the communications and data flow throughout the program. Note that in Figure 5, it is not obvious whether $MAIN$ is able to communicate with

```
41 int main(int argc, char** argv)
42 {
43   csp::Start_CPPCSP();
44   {
45     SHMChannel<int> x;
46     SHMChannel<int> xst;
47     LockChannel xm;
48     csp::Bucket xcv;
49     int lcl_x;
50     int lcl_xst;
51     int lcl_xm;
52     csp::Chanin<int>& x_in = x.reader();
53     csp::Chanout<int>& x_out = x.writer();
54     csp::Chanin<int>& xst_in = xst.reader();
55     csp::Chanout<int>& xst_out = xst.writer();
56     csp::Chanin<int>& xm_in = xm.reader();
57     csp::Chanout<int>& xm_out = xm.writer();
58     csp::ScopedForking * thr_1 =
59         new csp::ScopedForking();
60     lcl_x = 0;
61     x_out.write(&lcl_x);
62     lcl_xst = 0;
63     xst_out.write(&lcl_xst);
64     thr_1->fork(new thr1(x_in, x_out, xst_out,
65                          xm_in, xm_out, xcv));
66     xm_in.read(&lcl_xm);
67     xst_in.read(&lcl_xst);
68     if (lcl_xst != 1)
69     {
70         xm_out.write(&lcl_xm);
71         xcv.fallInto();
72         xm_in.read(&lcl_xm);
73     }
74     xm_out.write(&lcl_xm);
75     delete thr_1;
76     x_in.read(&lcl_x);
77     printf("%d\n", lcl_x);
78   }
79   csp::End_CPPCSP();
80   return 0;
81 }
```

Figure 4. The output of the basic transformation of the main function to a C++CSP process. Lines 58-77 make up the logic of the process, while the other lines consist of boilerplate declarations of channels and channel ends.

itself and whether $THR1$ is able to communicate with itself; neither are valid communications.

These invalid communications appear because of circular dependences in the network graph. Circular dependences occur when the same process reads from and writes to a channel. When circular dependences occur, threaded dependence analysis of the relevant processes is used to determine in which order (*i.e.*, read, then write; or write, then read) the channel is accessed by each process and which pairs of processes may actually communicate.

The **channel splitting** strategy can eliminate circular dependences. For a channel x, two channels are created, x_1 and x_2. In the process in which the read and write occurred, the instance of the channel being written to is transformed to x_1, as are all of the instances identified by data flow analysis as being able to see the write (*i.e.*, a forward slice to the next read). Likewise, the instance of the channel being read from is transformed to x_2, as is the backward slice from that point. Channel splitting enforces, in the network graph, the property that the process cannot be the originator of its own input. Figure 6 shows the effect of splitting the x channel.

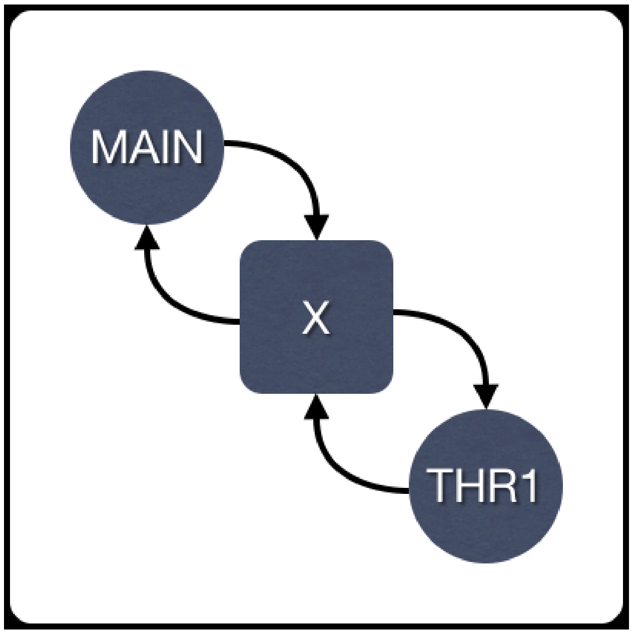

Figure 5. A visual depiction of the communications network between processes $MAIN$ and $THR1$, with respect to channel X.

B. Process Splitting

Circular dependences on the CSP network graph can also have the effect of making the order of process execution non-obvious. In the case of Figure 5, which of MAIN or THR1 executes first is non-obvious from the graph. Analysis of the relevant processes can determine for which of them is the initial event observable.

Such a circular dependence can be solved by **process splitting**, which involves breaking a process P into two processes P_1 and P_2. In this case, the $MAIN$ process is split into $MAIN_1$ and $MAIN_2$ between the write to x and the read from x, as shown in Figure 7. For any local variable in P defined in P_1 and on which a dependence exists in P_2, an additional one-to-one channel must be defined to transmit that value from P_1 to P_2.

When combined with the channel splitting strategy, process splitting is effective in removing most accidental

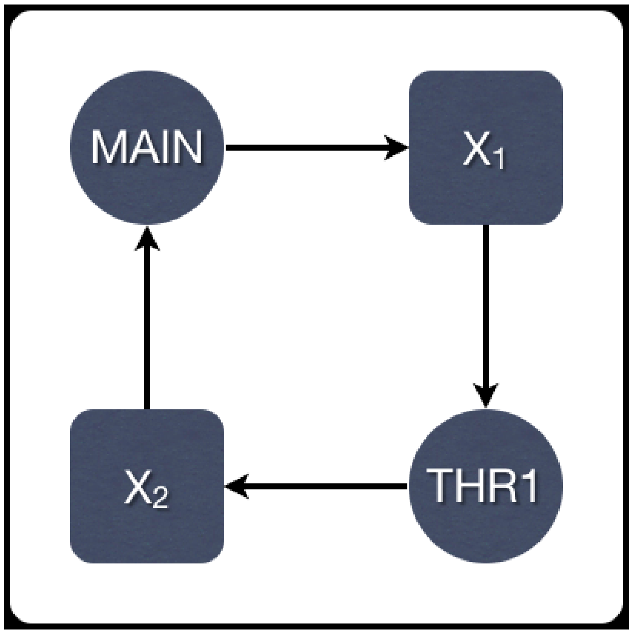

Figure 6. Channel X has been split into X_1 and X_2. Note how this makes the behavior of the program more clear than Figure 5.

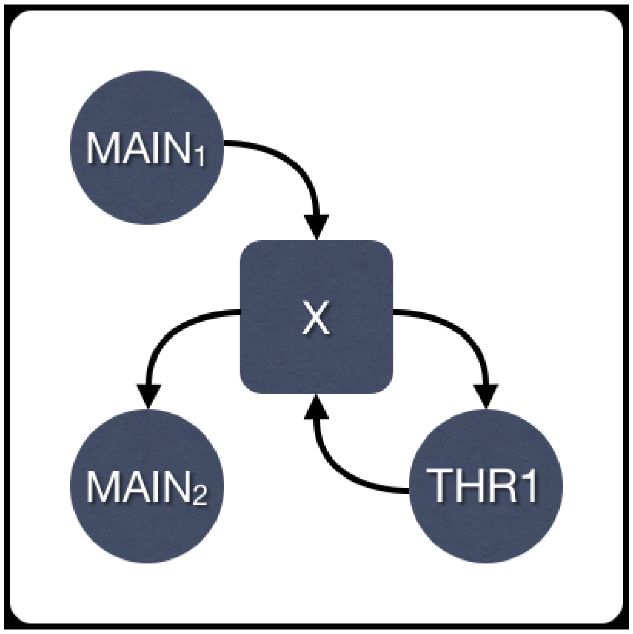

Figure 7. Process $MAIN$ has been split into processes $MAIN_1$ and $MAIN_2$. This helps to make the order of operations explicit from within the network graph.

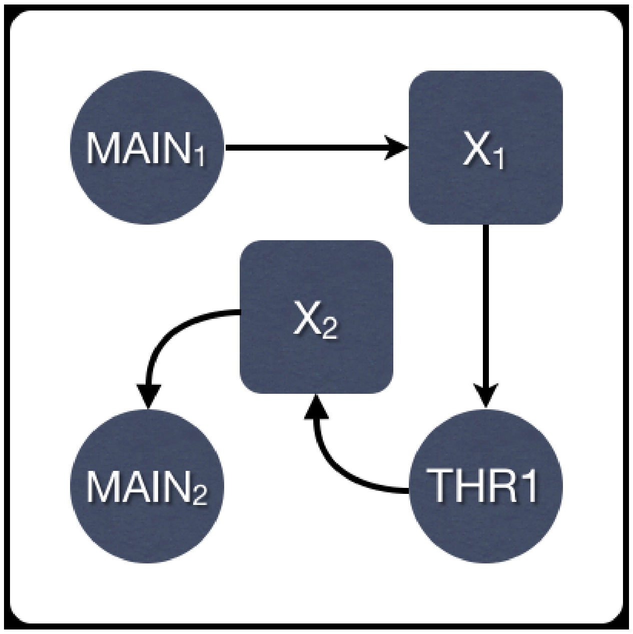

Figure 8. Channel splitting and process splitting combined can remove most ambiguity from process networks, in the absence of genuine circular data dependences.

circular dependences in a CSP network graph, leaving only genuine circular data dependences inherent to the program. The communications network in Figure 8 resembles the ideal CSP specification for the example program.

C. *SHMChannel Replacement*

The primary impediment to analysis of the naïvely converted program is the low-level nature of the SHMChannels, LockChannels, and buckets. The long-term goal of the *thr2csp* research is to develop methods for eliminating the use of locks and semaphores. This can be achieved only by converting SHMChannels into normal CSP channels. SHMChannels have the properties of asynchrony and persistence, while normal CSP channels are synchronous, and each read requires a write. This means that if a channel links two processes, then there must be exactly one write for every read. Such a relationship implies a much stronger pairing and ordering between communicating processes, than between processes sharing in an SHMChannel.

To replace SHMChannels algorithmically with normal CSP channels requires precise dependence analysis in order to build a model of the data flow into the process network. The precise data flow analysis of multithreaded programs is undecidable in general [8], so this process will never be fully automated. Nevertheless, dependence analysis can be used to provide conservative approximations of

```
class thr1 : public  csp::CSProcess
{
private:
  csp::Chanin<int> x1_in;
  csp::Chanout<int> x2_out;
protected:
  void run()
  {
    int lcl_x;
    x1_in.read(&lcl_x);
    lcl_x = lcl_x + 100;
    x2_out.write(&lcl_x);
  }
public:
  thr1(const csp::Chanin<int>& x1_in_,
       const csp::Chanout<int>& x2_out_)
  {
    x1_in = x1_in_;
    x2_out = x2_out_;
  }
};
```

Figure 9. The ideal `thr1` process in C++CSP

```
int main(int argc, char** argv)
{
  csp::Start_CPPCSP();
  {
    One2OneChannel<int> x1;
    One2OneChannel<int> x2;
    int lcl_x;
    csp::Chanin<int>& x1_in = x1.reader();
    csp::Chanout<int>& x1_out = x1.writer();
    csp::Chanin<int>& x2_in = x2.reader();
    csp::Chanout<int>& x2_out = x2.writer();
    csp::ScopedForking * thr_1 =
        new csp::ScopedForking();
    lcl_x = 0;
    x1_out.write(&lcl_x);
    thr_1->fork(new thr1(x1_in, x2_out));
    delete thr_1;
    x2_in.read(&lcl_x);
    printf("%d\n", lcl_x);
  }
  csp::End_CPPCSP();
  return 0;
}
```

Figure 10. The ideal `main` process in C++CSP

the data flow across multiple threads [9], [11], [12]. Thus, the current thrust of this research is in developing tool support to guide the user in the process of transforming accesses to SHMChannels into accesses to CSP channels.

Figures 9 and 10 show examples of the ideal conversion of SHMChannels to CSP channels in the example program. In this case, the xst variable has been identified as a concurrency-control variable unnecessary to the higher-level logic of the program, and has been eliminated. The splitting of the x channel, using knowledge derived from the analysis of data flow according to the mutexes and semaphores, has made the data flow explicit in the network graph. The x_1 and x_2 channels have been converted to normal CSP channels because there is a one-to-one relationship between reads and writes. The elimination or transformation of all SHMChannels into CSP channels with strong data-flow properties has enabled the elimination of the mutex and semaphore concurrency-control constructs.

V. Conclusions and Future Work

The work described in this paper demonstrates the possibility of converting a multithreaded program into a message-passing program built on top of the C++CSP library. Additional steps beyond the initial transformation improve the quality of the communications graph and render the resulting program more CSP-like. Unfortunately, this process is not and can never be fully automated due to the imprecision inherent in the analysis of multithreaded programs.

Significant additional work remains to be done regarding the implementation of the initial transformation from multithreaded code to C++CSP code. Currently, our handling of structures such as arrays, structs, and classes is naïve; such handling will have to be improved to guarantee no out-of-channel communication between the processes. Furthermore, the model of condition variables is currently unable to handle the `pthread_cond_timedwait` function, and the handling of all signals as broadcasts, while technically correct, is quite different from actual implementations of POSIX threading. Finally, all calls to C and C++ standard libraries must be wrapped to ensure that side-effects are not propagated between threads.

The primary thrust of the research now is to build a graphical development tool on top of Eclipse to allow the user to guide the process of converting SHMChannels into CSP channels. The program will display the (possibly imprecise) dependences between the threads, and give the user control over the "what-if" analysis of process and channel splitting.

It is our hope that the guided transformation of multithreaded programs into CSP programs will improve the ability to maintain and comprehend such systems, as well as demonstrate the feasibility of alternative concurrency mechanisms to shared-memory multithreading.

References

[1] D. Butenhof, *Programming With Posix Threads*. Addison-Wesley Professional, 1997.

[2] D. Lea, *Concurrent programming in Java*. Addison-Wesley Reading, Mass, 2000.

[3] E. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, 2006.

[4] C. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[5] A. Salcianu and M. Rinard, "Pointer and escape analysis for multithreaded programs," *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pp. 12–23, 2001.

[6] G. Naumovich and G. Avrunin, "A Conservative Data Flow Algorithm for Detecting All Pairs of Statements that May Happen in Parallel," 1998.

[7] G. Naumovich, G. Avrunin, and L. Clarke, "An Efficient Algorithm for Computing MHP Information for Concurrent Java Programs," *Software Engineering-Esec/Fse'99: 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 6-10, 1999: Proceedings*, 1999.

[8] G. Ramalingam, "Context-sensitive synchronization-sensitive analysis is undecidable," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 22, no. 2, pp. 416–430, 2000.

[9] J. Krinke, "Advanced Slicing of Sequential and Concurrent Programs," Ph.D. dissertation, Universität Passau, 2003.

[10] ——, "Context-sensitive slicing of concurrent programs," *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 178–187, 2003.

[11] M. Nanda and S. Ramesh, "Interprocedural slicing of multithreaded programs with applications to Java," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 28, no. 6, pp. 1088–1144, 2006.

[12] D. Giffhorn and C. Hammer, "An Evaluation of Slicing Algorithms for Concurrent Programs," *Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference on*, pp. 17–26, 2007.

[13] S. Brookes, C. Hoare, and A. Roscoe, "A theory of communicating sequential processes," *Journal of the ACM (JACM)*, vol. 31, no. 3, pp. 560–599, 1984.

[14] C. Hoare, "Communicating Sequential Processes," 1985.

[15] A. Roscoe, *The theory and practice of concurrency*. Prentice Hall, 1997.

[16] S. Schneider, *Concurrent and real-time systems: the CSP approach*. Wiley, 2000.

[17] A. Roscoe, "Model-checking CSP," *A Classical mind: essays in honour of CAR Hoare*, p. 353, 1994.

[18] N. Brown and P. Welch, "An Introduction to the Kent C++ CSP Library," *Communicating Process Architectures*, vol. 61, pp. 139–156, 2003.

[19] N. C. C. Brown, "C++CSP2: A Many-to-Many Threading Model for Multicore Architectures," in *Communicating Process Architectures 2007*, A. A. McEwan, W. Ifill, and P. H. Welch, Eds., jul 2007, pp. 183–205.

[20] E. Visser, "Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9," in *Domain-Specific Program Generation*, ser. Lecture Notes in Computer Science, C. Lengauer *et al.*, Eds. Spinger-Verlag, June 2004, vol. 3016, pp. 216–238. [Online]. Available: http://www.cs.uu.nl/research/techreps/UU-CS-2004-011.html

[21] P. Anderson and T. Teitelbaum, "Software inspection using codesurfer," in *Workshop on Inspection in Software Engineering (CAV 2001)*, 2001.

[22] P. Anderson, T. Reps, and T. Teitelbaum, "Design and implementation of a fine-grained software inspection tool," *IEEE Transactions on Software Engineering*, vol. 29, no. 8, pp. 721–733, 2003.