Proceedings of the Ninth International Workshop on Automated Verification of Critical Systems (AVOCS 2009)

Automatically Generating CSP Models for Communicating Haskell Processes

Neil C. C. Brown

12 pages

Automatically Generating CSP Models for Communicating Haskell Processes

Neil C. C. Brown[1]

[1] neil@twistedsquare.com, http://www.cs.kent.ac.uk/~nccb2/
Computing Laboratory, University of Kent, UK

Abstract: Tools such as FDR can check whether a CSP model of an implementation is a refinement of a given CSP specification. We present a technique for generating such CSP models of Haskell implementations that use the Communicating Haskell Processes library. Our technique avoids the need for a detailed semantics of the Haskell language, and requires only minimal program annotation. The generated CSP_M model can be checked for deadlock or refinements by FDR, allowing easy use of formal methods without the need to maintain a model of the program implementation alongside the program itself.

Keywords: CSP, Automatic Model Generation, Haskell

1 Introduction

Programs designed using formal methods such as Communicating Sequential Processes (CSP) [Hoa85, Ros97] typically have a specification and a more complicated implementation. A tool such as FDR [For97] can be used to check that the implementation is a refinement of the specification. However, determining the CSP model of an implementation – written in an executable programming language using a CSP-based library – can be difficult due to problems with unclear semantics, or translation errors. Generating a model from the program should thus be done automatically.

Generating a formal model of an existing program typically requires complete access to the program's source code (including libraries) and a detailed semantics of the programming language. Both requirements can be problematic: libraries may be closed source, and the programming language may lack a detailed semantics, especially a platform independent semantics that includes the semantics of the threading system and memory model.

Communicating Haskell Processes (CHP) [Bro08] is a library for the functional programming language Haskell with a strong correspondence to CSP. CHP allows for execution of CSP-like programs, combining the concurrency concepts of CSP with the expressive power of Haskell.

In this paper we describe a technique for generating formal models of CHP programs without the need for source code analysis. We take advantage of Haskell's purity and thus do not require a semantics of the Haskell language. We describe the class of programs that can be modelled, along with several examples. Our technique can be used to generate models to check for refinement of specifications, or for generating models of rapidly prototyped programs (for example, using agile development methods).

2 CHP

Although Haskell is a functional programming language, it has support for imperative programming through the concept of monads: a monad captures a common pattern that can be used for imperative programming. Thus, a CHP program can be conceptually separated into the pure functional computations, and the imperative communication aspects.

For example, this is a map process in CHP that transforms items as they pass through:

```
map :: (a -> b) -> Chanin a -> Chanout b -> CHP ()
map f input output = do x <- readChannel input
                        writeChannel output (f x)
                        map f input output
```

The top line is the type signature of the process. Types that begin with a lower-case letter are parameterised types – thus this type reads: given a function from some type a to some type b, an incoming channel of type a and an outgoing channel of type b, the process will yield a CHP process with the `()` unit type as its return.

The implementation of the process is straightforward. A value is read from the input channel and bound to the name x. The result of applying the function f to x is sent on the output channel, and then the process recurses. It can be seen that the behaviour of the pure function f has no effect on the communication behaviour of the process (providing that f terminates).

3 Approach

Our approach is to take the CHP library and provide a mirror implementation with near-identical Application Programming Interface (API). This mirror implementation does not properly execute the code as the original CHP would, but instead traces[1] the structure of the program and produces a CSP model of the program.

The original library has API functions for creating channels and barriers, and communicating and synchronising on them (see section 7.1 for details on the synchronisation semantics). The mirror version has the same creation functions, but the internal definition of channels and barriers changes so that they simply hold a unique identifier. The communication and synchronisation functions are changed in the mirrored version to record the synchronisation in a CSP model, rather than performing an actual synchronisation.

The mirror definitions for the parallel composition operator and the external choice operator generate the models for all processes involved, and then compose the models using parallel composition and external choice. The monad system in Haskell means that we can alter the definition of sequential composition (akin to overloading the semi-colon in C++ or Java) to compose the models in sequence. Thus only the pure computations might be executed[2], but none of the imperative side-effecting parts of the process. This is because the latter are replaced by code that only generates the model. The three types of imperative statements that can occur in the CHP monad are:

[1] The sense of tracing used here is not related to the idea of traces in CSP.

[2] Because of Haskell's laziness, these will only be executed if the result is required to make a choice about the program's flow (see section 3.1 for more details).

1. Channel/barrier creation,

2. Channel communication and barrier synchronisation, and

3. Lifted IO actions (e.g. writing to a file, opening a network socket).

We have already explained the first two; IO actions are described in section 5. Given the existence of this mirror library, the main change needed to generate a model rather than run a program is to change the normal *import Control.Concurrent.CHP* – perhaps using a pre-processor for ease of use – to *import Control.Concurrent.CHP.Model*.

This library substitution approach avoids the need for source code analysis, which means that no source code is used by the tool. One drawback to this is that looping and recursive behaviours cannot be detected automatically, and require extra annotations (see section 4).

3.1 Reading from Channels

Our approach benefits from Haskell's pure and lazy nature. Many processes, such as the earlier *map* process, have behaviour that is unaffected by the values being transmitted. It is thus possible to return the value $\perp$ from reading a channel, i.e. a value that will give an error if evaluated. The consequences of this change are as follows.

- If the value read from the channel is discarded, returning $\perp$ will not have any adverse effect.

- If the value is sent on another channel, this will also not cause an error as we re-implement the channel output function. We are able to track the different $\perp$ values throughout the system to know which channel the value came from.

- If the value is used as part of another computation that does not end up being evaluated (e.g. because the new value is sent down a channel, or is not needed), there will again not be a problem because of Haskell's laziness.

- If the value is evaluated because it is used to determine the program's flow, this will cause the error from the $\perp$ value to occur. This is explored further in section 5.

3.2 Value Sources

The separation of monadic side effects from pure code means that any value in a CHP process must come from one (or a combination) of three sources:

1. A deterministic source, e.g. a constant in the program, or a pure computation with constant inputs.

2. A value read from a communication channel, or returned from an IO action.

3. A parameter to the process.

Values in the first category are the same every time the program is run, and thus any decisions about the program's behaviour based solely on these values will always have the same outcome. Values in the second category are dealt with in section 5, while values in the third category are explained in section 4.

It should be emphasised that our approach is not the same as producing a model based on observation of executing the process. The combination of the replacement of the monad, and the use of $\perp$ bottom values means that we can ensure that the model we generate is an accurate reflection of the process's behaviour in *every* execution, with the exception of issues dealt with in later sections of this paper, and known Haskell "back doors" such as the aptly named *unsafePerformIO*.

4 Recursion and Looping

Most processes have infinite behaviour, via the use of recursion. A very straight-forward example is a process that consumes input:

```
blackHole :: Chanin a -> CHP ()
blackHole input = do readChannel input
                     blackHole input
```

This can also be written using the common Haskell function *forever* that infinitely repeats in sequence a monadic action:

```
blackHole' :: Chanin a -> CHP ()
blackHole' input = forever (readChannel input)
```

Infinite behaviour cannot be detected with our substituted monad; there is no way to determine after one million consecutive inputs whether the next statement will definitely be a input. We must require the programmer to add an annotation to aid in spotting recursion.

The *process* annotation is placed as follows:

```
blackHole :: Chanin a -> CHP ()
blackHole = process "blackHole" inner
  where
    inner input = do readChannel input
                     blackHole input
```

The *process* annotation uses Haskell's type-classes to take as its second argument a process with N arguments, and return a process that takes the same N arguments. The behaviour of the *process* wrapper function is to record the arguments to the process ready for future equality checking.

When *blackHole* is first called with a channel c, the *process* annotation stores a pair of the process name and a list containing all the arguments (in this case, just c) as the key in an associative map with a placeholder for the value. When the process then recurses, the process name and list of arguments (which is again, the singleton list with c) is looked up in the map. A value is found, which causes the execution of the process to return the placeholder. Thus only one execution of the process is examined, and the recursive call is skipped.

The analysis of the *blackHole* process is then complete, and the placeholder is substituted for a definition of the process's behaviour. This relies on the user-provided guarantee that all parameters that affect a process's behaviour are included in the argument list; a process may not use any other free names.

As well as the *process* annotation, we supply a replacement for the Haskell library function *forever*, named *foreverP* that has identical semantics, but correctly deals with the repeating behaviour in our mirror implementation. These annotations can be used with the normal library (*process* becomes benign, *foreverP* is simply defined as *forever*) without effect, but then they take on significance in the mirror implementation.

5 IO Actions

An interaction with the program's external environment is an action in the IO monad in Haskell. This can include (but is not limited to) interactions with files, sockets and graphical user interfaces. Results of these interactions cannot be predicted and are non-deterministic. Our generated model accounts for this by modelling the interactions as events selected by the environment. The complication is modelling the value returned by operations such as reading from a file.

With source code analysis it would be possible to analyse how the result affects the program's subsequent behaviour. However, with our approach, the program's behaviour can only be determined by testing it with each possible value. For values with small finite domains, such as booleans, or enumerated types, this is easily achieveable. For values such as 32-bit integers, it becomes infeasible, and for values with infinite domains such as strings, it is impossible.

Our analysis of a program is therefore accurate and feasible if the IO actions return values with small finite domains, or if the values are discarded. This is often the case for IO actions: for example, printing to the screen, waiting for a specified amount of time or writing to a file all return values of the unit type, which has a domain size of one (which also means the value is not usually inspected). We can extend our approach to cover some other cases where the value is used and the domain is large; the problem of determining a program's behaviour for a range of inputs without source code analysis is identical to the problem of program testing. Several developments have been made in this field in Haskell.

Claessen and Hughes' QuickCheck generated random values of a given type and tested that specified properties held on the output [CH00]. This was made more systematic in Runciman et al.'s SmallCheck, which generated all values up to a given depth [RNL08]. The depth of a list would be its length, whereas the depth of an integer would be its absolute value. The clever extension, Lazy SmallCheck, took advantage of Haskell's laziness to specialise on demand, allowing more efficient exploration of the state space.

We use the Lazy SmallCheck library to explore a program's behaviour given the result of an IO action. Crucially, Lazy SmallCheck will efficiently detect if the value is not used to affect the program's behaviour, as the first value it will try will be $\perp$. If the program does not raise an error with this value, it must not have evaluated it, and thus it cannot have altered its behaviour based on the value.

We use Lazy SmallCheck with an arbitrary maximum depth. If the deepest value is not reached in the lazy search, we can deduce that we have covered all possible behaviours of the program.

If this is not the case, the model generated is only an approximation of the program's behaviour, and much of the advantages of the formal method are lost. This is a necessary restriction of our approach and one that we expose to the user. We deal with values returned from reading channels in the same manner as the result of IO actions.

6 Examples

We provide two examples of our system: the first is a small example demonstrating refinement checking, the second is a larger example demonstrating deadlock freedom.

6.1 Copy Buffer

A simple example of a refinement check, taken from the FDR manual [For97], is that of a copying process, specified as follows:

$$COPY = left?x \rightarrow right!x \rightarrow COPY$$

The example implementation given in the manual, in CSP, is:

$$SEND = left?x \rightarrow mid!x \rightarrow ack \rightarrow SEND$$
$$REC = mid?x \rightarrow right!x \rightarrow ack \rightarrow REC$$
$$SYSTEM = (SEND \underset{\{X\}}{||} REC) \setminus X \text{ where } X = \{mid, ack\}$$

We can create the analogue of this implementation in CHP as follows:

```
system :: forall a. Typeable a => Chanin a -> Chanout a -> CHP ()
system input output
  = do c <- newChannelWithLabel "mid"
       d <- newBarrierWithLabel "ack"
       enroll d (\d0 -> enroll d (\d1 ->
         send input (writer c) d0 <|> rec output (reader c) d1))
       return ()
  where
    send :: Chanin a -> Chanout a -> EnrolledBarrier -> CHP ()
    send = process "send" (\input mid ack ->
      do x <- readChannel input
         writeChannel mid x
         syncBarrier ack
         send input mid ack)

    rec :: Chanout a -> Chanin a -> EnrolledBarrier -> CHP ()
    rec = process "rec" (\output mid ack ->
      do x <- readChannel mid
         writeChannel output x
         syncBarrier ack x
         rec output mid ack)
```

The backslash represents a lambda, and introduces an anonymous function (e.g. `\x -> x` is the identity function). The *enroll* function takes as its arguments a barrier, and a function that itself takes an enrolled barrier and yields a CHP process. This higher-order process style is used to scope enrolling on and resigning from the barrier.

This program can be executed normally with the CHP monad as part of a larger system, or used with our new mirror library to generate a model, which results in the following:

```
channel ack
channel in
channel mid
channel out
main_0= (((send_1) [|{| ack , mid |}|] (rec_2)))
rec_2= (mid?x_2 -> out!x_2 -> ack -> rec_2)
send_1= (in?x_1 -> mid!x_1 -> ack -> send_1)
main = main_0
```

It can be seen that, a few spurious brackets and appended unique identifiers aside, this is the same as the CSP we began with – except for the hiding. We can now prove this is a refinement of the original specification by adding the following lines to the FDR script:

```
COPY = in?x -> out!x -> COPY
assert COPY [FD= (main\{mid,ack})
assert COPY [T= (main\{mid,ack})
```

These refinements check successfully. We were able to take a specification, implement a CHP equivalent (in this case we had some CSP for the implementation, but this was not necessary), and make a successful refinement check against the original specification.

The main manual step required was that we hid the `mid` and `ack` events. We could attempt to infer when to hide events (a difficult option), or we could add an operator in the code to hide events. For example, we could modify a line of our system process to be:

```
enroll d (\d0 -> enroll d (\d1 ->
  send input (writer c) d0 <|> rec output (reader c) d1
    <\> [c, d]))
```

6.2 Dining Philosophers

As an example of generating a CSP model from a CHP program and checking for deadlock, we use the dining philosophers problem. The code for the deadlocking dining philosophers, that can be executed normally, or used to produce traces [BS08], is as follows:

```
fork :: EnrolledBarrier -> EnrolledBarrier -> CHP ()
fork = process "fork" (\left right ->
    foreverP ((do syncBarrier left
                  syncBarrier left )
              <-> (do syncBarrier right
                      syncBarrier right )))
```

```
philosopher :: EnrolledBarrier -> EnrolledBarrier -> CHP ()
philosopher = process "philosopher" (\left right ->
  foreverP (
   do randomDelay
      syncBarrier left <|> syncBarrier right
      randomDelay
      syncBarrier left <|> syncBarrier right ))
  where
    randomDelay :: CHP ()
    randomDelay = liftCHP $ ( liftIO $ getStdRandom (randomR (500000, 1000000))) >>= waitFor

college :: Int -> CHP ()
college = process "college" (\nPhil ->
  withBarrierPairListWithStem nPhil "fork_left_phil" (\forkLeftChans ->
  withBarrierPairListWithStem nPhil "fork_right_phil" (\forkRightChans ->
  runParallel_ (
    [fork (fst (forkRightChans !! n)) (fst (forkLeftChans !! ((n + 1) `mod` nPhil)))
    | n <- [0 .. nPhil - 1]]
    ++
    [philosopher (snd (forkLeftChans !! n)) (snd (forkRightChans !! n))
    | n <- [0 .. nPhil - 1]]))))
```

The *process* annotations on the fork and philosopher are not strictly necessary (the use of *foreverP* catches the infinite behaviour) but help to label the processes in the generated model. The `<->` operator is external choice, while `<|>` is parallel composition (and *runParallel_* is a list version).

This real running code can then be changed to use the CHP-model library by changing a single import statement (omitted for brevity). The generated model for three philosophers is shown in figure 1. If we append the line `assert main :[ deadlock free ]` to that script, FDR produces a trace of one of the deadlocks in the system:

```
BEGIN TRACE example=0 process=0
fork_right_phil2
fork_right_phil1
fork_right_phil0
END TRACE example=0 process=0
```

7 Post-Processing

After the model for a program has been determined, it is a relatively simple matter to print it out in the machine-readable CSP_M form that the FDR model checker expects. The only current drawback is that each different combination of arguments to an annotated process will correspond to a different process in the generated output. Returning to our example of the input consuming process, the system with *blackHole c* `<|>` *blackHole d* generates:

```
channel c
channel d
blackHole_1= (c?x_1 -> blackHole_1)
```

```
blackHole_2= (d?x_2 -> blackHole_2)
main_0= (((blackHole_1) ||| (blackHole_2)))
main = main_0
```

Even though the behaviour of the two black hole processes is identical except for the channel involved, we currently generate two instantiations of the process rather than one parameterised process. In future we would like to maintain the correctness of the specification, but also reduce its verbosity by merging such processes together.

7.1 Alphabets

In CHP, the synchronisation rules are as follows. A channel requires exactly two processes to synchronise on it. A barrier has an enrollment count, and a number of processes equal to that count must synchronise on the barrier for it to complete.

In Roscoe's CSP [Ros97], an event can have any number of parties, from one upwards. Which processes must synchronise with each other on an event is determined by the shared alphabet when the two processes are composed in parallel. Given this CSP:

$$P = a \rightarrow b \rightarrow c \rightarrow \text{SKIP}$$
$$Q = b \rightarrow \text{SKIP}$$
$$R = c \rightarrow \text{SKIP}$$
$$ALL = P \underset{\{b\}}{\|} Q \underset{\{c\}}{\|} R$$

The event a will involve just P, whereas b will involve P and Q, and c will involve P and R. Should the event a have been included in the alphabet of either parallel composition, P would cause deadlock as the other process in the composition would not be offering the event a.

In translating CHP programs into CSP models, we must infer the alphabets for parallel compositions. Since all of our events have unique identifiers we are able to follow a simple rule: the events in the alphabet of parallel composition are the intersection of the sets of events engaged in by the two processes being composed.

This rule works correctly for most programs (including the dining philosophers example). However, consider the following program:

```
p = do a <- newBarrierWithLabel "a"
       b <- newBarrierWithLabel "b"
       enroll a (\a0 -> enroll a (\a1 -> enroll b (\b0 -> enroll b (\b1 ->
           runParallel [do syncBarrier a0
                           syncBarrier b0
                       , syncBarrier a1 ])))
```

This program will deadlock when run, as only one process (of two enrolled) will synchronise on the barrier b. If the specification is generated with our original simple rule, we get this specification:

```
p = (a -> b -> SKIP) [|{| a |}|] (a -> SKIP)
```

This specification will not deadlock, as the event *b* is not in the shared alphabet with any other process. More generally, if less processes are using a barrier than are enrolled, or if only one process is using a particular channel, the model generated with our simple alphabet rule will be incorrect.

The simplest solution to this is as follows. We augment our framework to track how many processes should be using a particular event (two for channels, the enrollment count for barriers). If the actual number of processes turns out to be lower than this, we compose the relevant processes in parallel with a dummy process (`SKIP`) with the events shared, for example:

```
p = ((a -> b -> SKIP) [|{| a |}|] (a -> SKIP))
      [|{| b |}|] SKIP
```

This new model will reveal the deadlock in the original program.

8 Summary

Our approach is able to generate models for the following features of CHP programs:

- sequential and parallel composition,
- external choice,
- barrier creation and synchronisations (but not a proper semantics of dynamic enrollment and resignation),
- channel outputs,
- channel inputs and lifted IO actions (where the result either has a small finite domain, or is not used to make a choice about the program's behaviour), and
- terminating pure computations.

The main area not supported is where the domain of channel-reads and IO-actions is large *and* the result is used to influence decisions about the program's flow. We also do not support programs with pure computations that do not terminate – such non-termination would also be problematic in the real running version of CHP.

9 Conclusions

We have demonstrated a technique for generating CSP models of Haskell programs that use the CHP library. It does not require complete access to the source code, and requires minimal program annotation. The technique can be used to generate models for prototype programs (or programs developed with an agile methodology) or to generate models for programs and perform a refinement check against a specification.

The CSP_M that is generated by our approach can be passed directly to tools such as FDR in order to prove properties such as freedom from deadlock or perform refinement checks. It can

also be used with other model-checkers (such as ProB [LF08]) or tools (such as FDR explorer [FW09]) on the specification. Alternatively, a tool such as ProBE [Ros97] could be used to explore the possible traces of a program by deciding which of all the available events should happen next.

9.1 Availability

The CHP library is already available for general use – more details can be found at http://www.cs.kent.ac.uk/projects/ofa/chp/. We hope to soon release the mirror implementation described in this paper that generates CSP models.

Bibliography

[Bro08] N. C. C. Brown. Communicating Haskell Processes: Composable Explicit Concurrency Using Monads. In *Communicating Process Architectures 2008*. Pp. 67–84. Sept. 2008.

[BS08] N. C. C. Brown, M. L. Smith. Representation and Implementation of CSP and VCR Traces. In *Communicating Process Architectures 2008*. Pp. 329–345. Sept. 2008.

[CH00] K. Claessen, J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proc. of International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, 2000.

[For97] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 Manual*. 1997.

[FW09] L. Freitas, J. Woodcock. FDR Explorer. *Formal Aspects of Computing* 21:133–154, 2009.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. http://www.usingcsp.com/

[LF08] M. Leuschel, M. Fontaine. Probing the Depths of CSP-M: A new FDR-compliant Validation Tool. *ICFEM 2008*, 2008.

[RNL08] C. Runciman, M. Naylor, F. Lindblad. Smallcheck and Lazy Smallcheck: automatic exhaustive testing for small values. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*. Pp. 37–48. ACM, 2008.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997. http://www.comlab.ox.ac.uk/people/bill.roscoe/publications/68b.pdf

```
channel fork_left_phil0
channel fork_left_phil1
channel fork_left_phil2
channel fork_right_phil0
channel fork_right_phil1
channel fork_right_phil2
college_1=
  (((fork_2)
    [|{| fork_left_phil1 , fork_right_phil0 |}|]
    ((fork_3)
     [|{| fork_left_phil2 , fork_right_phil1 |}|]
     ((fork_4)
      [|{| fork_left_phil0 , fork_right_phil2 |}|]
      ((philosopher_5) ||| ((philosopher_6) ||| (philosopher_7)))))))
fork_2= (repeated_8)
fork_3= (repeated_9)
fork_4= (repeated_10)
main_0= (college_1)
philosopher_5= (repeated_11)
philosopher_6= (repeated_12)
philosopher_7= (repeated_13)
repeated_8=
  (((fork_right_phil0 -> fork_right_phil0 -> SKIP)
    []
    (fork_left_phil1 -> fork_left_phil1 -> SKIP))
   ; repeated_8)
repeated_9=
  (((fork_right_phil1 -> fork_right_phil1 -> SKIP)
    []
    (fork_left_phil2 -> fork_left_phil2 -> SKIP))
   ; repeated_9)
repeated_10=
  (((fork_right_phil2 -> fork_right_phil2 -> SKIP)
    []
    (fork_left_phil0 -> fork_left_phil0 -> SKIP))
   ; repeated_10)
repeated_11=
  (((fork_left_phil0 -> SKIP) ||| (fork_right_phil0 -> SKIP))
   ; ((fork_left_phil0 -> SKIP) ||| (fork_right_phil0 -> SKIP))
   ; repeated_11)
repeated_12=
  (((fork_left_phil1 -> SKIP) ||| (fork_right_phil1 -> SKIP))
   ; ((fork_left_phil1 -> SKIP) ||| (fork_right_phil1 -> SKIP))
   ; repeated_12)
repeated_13=
  (((fork_left_phil2 -> SKIP) ||| (fork_right_phil2 -> SKIP))
   ; ((fork_left_phil2 -> SKIP) ||| (fork_right_phil2 -> SKIP))
   ; repeated_13)
main = main_0
```

Figure 1: The generated model for the deadlocking version of the dining philosophers with three philosophers.