Verifying the Uniform Candy Distribution Puzzle with CSP-Prover

Yoshinao Isobe
National Institute of AIST, Japan
y-isobe@aist.go.jp

Markus Roggenbach
Swansea University, United Kingdom
M.Roggenbach@Swan.ac.uk

Abstract

In this paper we formally verify that the Uniform Candy Distribution Puzzle is self-stabilizing: Given a valid start configuration, eventually the Puzzle will evolve to a 'stable' situation in which it will remain. In terms of verification, the Uniform Candy Distribution Puzzle forms a scalable parametrized distributed system: The Puzzle comes in various sizes, for each size of the Puzzle there are infinitely many valid start configurations, the Puzzle evolves following local rule applications. We describe how to model the Uniform Candy Distribution Puzzle in the process algebra CSP*, give a mathematical argument for its self-stabilizing property, and formalize the proof with the interactive theorem prover* CSP-*Prover.*

1 Introduction

The "Uniform Candy Distribution Puzzle"[1] is a classical example of a self-organizing distributed system:

> **Uniform Candy Distribution Puzzle**:
>
> k children are sitting in a circle (see Fig.1). Each child starts out with an even number of candies. The following step is repeated indefinitely: every child passes half of her/his candies to the child on her/his left; any child who ends up with an odd number of candies is given another candy by the teacher.

One might think that the teacher may keep handing out more and more candies indefinitely. However, this is not true. Eventually the teacher will stop handing out candies and, in fact, the following holds:

> **Claim:** Eventually every child will hold the same number of candies.

[1] It appears to be impossible to identify the inventor of the puzzle, one reference, however, is [2].

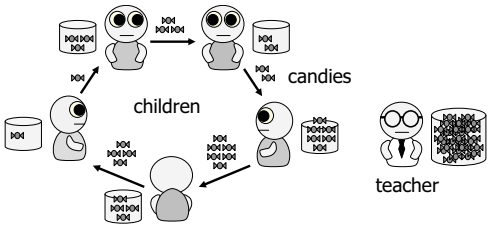

Figure 1. Uniform Candy Distribution Puzzle

Assuming this claim to be correct, the Uniform Candy Distribution Puzzle exhibits typical properties of a self-organizing system, see e.g. [4]:

- The puzzle evolves over time without influence from the outside.
- The state space of the puzzle reduces over time, it has an equal distribution of candies as an attractor.
- The children and the teacher act locally only.

As intuitive as the above description of our puzzle appears at first sight, it leaves many questions open when it comes to the notion of a "step": How does the exchange of candies happen? What happens if there is an selfish child who wants to get her/his new candies before this very child is willing to pass over her/his candies to the left? Is the order important, in which the teacher gives out candies to the children?

The process algebra CSP, see e.g. [1, 5, 16, 17], captures distributed systems in a precise way. In CSP, processes make progress on their own, the exchange of messages serves as a synchronization mechanism. For verification, the CSP approach is to model both, the distributed system as well as a desired property as CSP processes. A system *Sys* has a property *Prop*, if the system is a refinement of the property, i.e., if the relation $Prop \sqsubseteq Sys$ holds (where $\sqsubseteq$ denotes the CSP refinement relation). In the proof of such a statement the process algebraic laws of CSP play a vital role: Thanks to completeness results, see e.g. [9, 16],

normally refinement statements can be proven by applying process algebraic laws solely.

When verifying a system, the refinement proofs often involve tasks where one would hope for tool support. This includes especially the tedious and therefore error-prone task of book-keeping on all the still open proof obligations and the repeated application of standard proof patterns. Here, the tool CSP-Prover [7, 8, 9, 10, 11] can be of assistance. On the technical side, CSP-Prover provides a deep encoding of CSP in the generic theorem prover Isabelle/HOL [14]. On the practical side, CSP-Prover offers its user a vast amount of proof infrastructure in the form of process algebraic laws and specialized tactics.

In the following, we show how to model the Uniform Candy Distribution Puzzle as a concurrent process in CSP, and how to encode this process within the input language of CSP-Prover. We then discuss why the analysis of this Puzzle presents a challenge and explain how CSP-Prover can assist in the proofs involved.

2 Modelling and Encoding

In this section, we explain how one can model the Uniform Candy Distribution Puzzle in CSP and how this model is then encoded in CSP-Prover.

2.1 Modelling the puzzle in CSP

First we reflect on the behaviour of the individual children. The activity of a single child can be seen as the transition graph shown in Fig. 2. This graph consists of three states $\texttt{Child}(n)$, $\texttt{ChildL}(n, x)$, and $\texttt{ChildR}(n)$, where n represents the number of candies a child holds, and x stands for the number of candies a child has received in the last move. The function *fill* is defined as

$$\mathit{fill}(n) = \texttt{ if } (\mathit{even}(n)) \texttt{ then } n \texttt{ else } (n+1).$$

The event $\mathit{left}!(n/2)$ means to send the value $(n/2)$ to the left, the event $\mathit{right}?x$ means to receive a number from the right and to bind the variable x to the received number. Starting in state $\texttt{Child(n)}$, a child has the option to send half of her/his candies to the left child and to go over to state $\texttt{ChildR}(n/2)$. Then, it waits to receives an unknown number of x candies from the right child and goes over into state $\texttt{Child(fill(n/2 + x))}$. If the new number $(n/2 + x)$ of candies is odd, the function *fill* adds 1 to this number. This corresponds to the teacher's supply. Alternatively, from the initial state $\texttt{Child(n)}$ the child also has the option to first receive x candies, which leads to state $\texttt{ChildL}(n, x)$. Then, the child passes over $n/2$ candies, and goes over to the state $\texttt{Child(fill(n/2 + x))}$.

The above discussed transition graph can be modelled in CSP as follows:

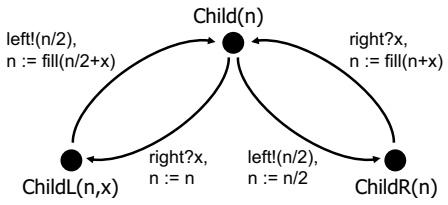

Figure 2. Transition graph of each child

$$\begin{aligned}
\texttt{Child}(n) &= (\mathit{left}!(n/2) \rightarrow \texttt{ChildR}(n/2)) \Box \\
&\quad\ (\mathit{right}?x \rightarrow \texttt{ChildL}(n, x)) \\
\texttt{ChildL}(n, x) &= \mathit{left}!(n/2) \rightarrow \texttt{Child}(\mathit{fill}(n/2 + x)) \\
\texttt{ChildR}(n) &= \mathit{right}?x \rightarrow \texttt{Child}(\mathit{fill}(n + x))
\end{aligned}$$

Here, $\rightarrow$ and $\Box$ are the prefix operator and the external choice operator of CSP, respectively. Intuitively, $a \rightarrow P$ is a process which can perform the event a and thereafter behaves like P. The process $P \Box Q$ behaves either like P or like Q depending on the initial actions. Here, the choice, which branch to take, is determined by events from the environment. The process definition of `Child`, `ChildL`, and `ChildR` consists essentially of infinitely many equations, one for each instance of the parameters n and x. In CSP, elements such as *left* and *right* for passing data are called *channels*, and elements such as `Child` for defining recursive behaviour are called *process names*.

In order to connect k `Child` processes into a circle $(k > 1)$, we carry out two steps: (1) we define a concurrent process $\texttt{LineCh}(\langle n_2, n_3, \ldots, n_k \rangle)$, which is the connection of $(k-1)$ children in one line. Here, the right hand of the i th child is connected to the left hand of the $(i+1)$ th child for $i = 2, \ldots, k-1$. (2) we define a concurrent process $\texttt{CircCh}(\langle n_1, n_2, \ldots, n_k \rangle)$, which is a circular connection of k children. Here, the right hand of the first child is connected to the left hand of the second child, and the left hand of the first child is connected to the right hand of the last (k th) child. In both steps we use the following notions: n_i denotes number of candies the i th child holds; $\langle n_1, \ldots, n_k \rangle$ is a list of even numbers $n_1, \ldots, n_k$.

These processes are inductively defined in CSP as follows:

$$\begin{aligned}
\texttt{LineCh}(\langle n \rangle) &= \texttt{Child}(n) \\
\texttt{LineCh}(\langle n \rangle \frown s) &= (\texttt{Child}(n) \longleftrightarrow \texttt{LineCh}(s)) \\
\texttt{CircCh}(\langle n \rangle \frown s) &= (\texttt{Child}(n) \Longleftrightarrow \texttt{LineCh}(s))
\end{aligned}$$

Here, $\frown$ is the concatenation operator of lists and the operators $\longleftrightarrow$ and $\Longleftrightarrow$ are defined from basic operators as follows:[2]

$$\begin{aligned}
P \longleftrightarrow Q &= (P[\![\mathit{right} \leftrightarrow \mathit{mid}]\!] \ |[\mathit{mid}]| \ Q[\![\mathit{left} \leftrightarrow \mathit{mid}]\!]) \backslash \mathit{mid} \\
P \Longleftrightarrow Q &= (P \ |[\mathit{left}, \mathit{right}]| \ Q[\![\mathit{left} \leftrightarrow \mathit{right}]\!]) \backslash \mathit{right}
\end{aligned}$$

[2]For simplicity, we follow in this paper the established CSP conventions. For example, the operator $|[\mathit{mid}]|$ expands to $|[\{\mathit{mid}(n) \mid n \in Nat\}]|$.

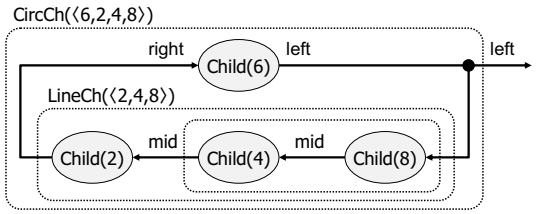

Figure 3. Structure of `CircCh`$(\langle 6,2,4,8\rangle)$

Here, $[\![a{\leftrightarrow}b]\!]$ is the renaming operator for exchanging channel-names a and b (note: if b is new then it works as renaming a to b), $|[X]|$ is the parallel composition for performing processes in parallel but synchronising events in the set X, and $\backslash X$ is the hiding operator for hiding events in the set X from other processes. Thus, $P{\longleftrightarrow}Q$ is a process obtained by renaming both of *left* in P and *right* in Q to a new common name *mid*, composing the renamed processes via *mid*, and finally hiding the shared channel *mid*. By hiding *mid*, we can repeatedly use *mid* for connecting children without conflicts. $P{\Longleftrightarrow}Q$ is similar to $P{\longleftrightarrow}Q$ except that both hands are connected and only the right hand is hidden. This means that we can observe the number of candies the first child has via the channel *left*. For example, Fig. 3 shows the structure of the process `CircCh`$(\langle 6,2,4,8\rangle)$.

2.2 Encoding the puzzle in CSP-Prover

It is easy to encode CSP-processes into CSP-Prover. Fig. 4 shows the encoded process `CircCh`, where some definitions such as `<---->` have been omitted. The complete code is available at CSP-Prover's web-site [7].

In Fig. 4, lines 1 and 2 define the type `Event` of events and the type `PN` of process names, respectively. These types are used for defining processes whose type is `(PN,Event) proc`. The parametrized type `proc` is provided by CSP-Prover. Next, lines 4–11 define the function `PNdef` which maps process names of type `PN` to processes of type `(PN,Event) proc`. There is a definition for each process name. CSP-Prover syntax is nearly the same as CSP syntax – except that conventional symbols such as $\Box$ are replaced by ASCII symbols such as `[+]`, and `$` is attached to each process name as a type conversion from a process name to a process. Isabelle's `recdef` mechanism for defining recursive functions makes pattern-matching of arguments available as shown in lines 6–11. When defining a recursive function by `recdef`, Isabelle requires a measure to guarantee termination of the defined function. For example, line 19 takes the length of lists as the measure; since `PNdef` is non-recursive, the empty set $\{\}$ (see line 5) can serve as measure. Note that line 13 declares the function `PNdef` to be the function `PNfun`. `PNfun` has the status of a reserved word of CSP-Prover and is automatically applied for unfolding process names. Finally, The processes `LineCh` and `CircCh` are defined as explained above in lines 19–24.

3 Verification

In this section we verify our claim that the Uniform Candy Distribution Puzzle has an attractor. To this end, we first present a proof for a synchronous version of the puzzle – which we then generalize in CSP-Prover to a proof on the asynchronous puzzle.

3.1 A known solution

Solutions of the Uniform Candy Distribution Puzzle have already been given, e.g. in the web-page [2]. One solution is as follows: Let s be the list of even numbers of the number of candies which the children hold. Then, after every child passed half of her/his candies to her/his left child and the teacher supplied candies if needed, the new list is given by the function $circNext(s)$, which is defined using the function $lineNext(s,x)$:

$$\begin{aligned} lineNext(\langle\rangle, x) &= \langle\rangle \\ lineNext(\langle n\rangle, x) &= \langle fill(n/2+x)\rangle \\ lineNext(\langle n,m\rangle^\frown s, x) &= \langle fill(n/2+m/2)\rangle \\ &\quad {}^\frown lineNext(\langle m\rangle^\frown s, x) \\ circNext(\langle\rangle) &= \langle\rangle \\ circNext(\langle n\rangle^\frown s) &= lineNext(\langle n\rangle^\frown s, n/2) \end{aligned}$$

In $lineNext(s,x)$, the variable x represents the number of candies to be passed to the last child. Since the children are sitting in a circle, in the function $circNext(\langle n\rangle^\frown s)$ this number x is half of candies of the first child, namely $n/2$. Take for example

$$\begin{aligned} circNext(\langle 4,2,10\rangle) &= \langle fill(2+1), fill(1+5), fill(5+2)\rangle \\ &= \langle 4,6,8\rangle. \end{aligned}$$

With these notations, the following properties hold: applying the function $circNext$ to a list s of even numbers (1) the maximum in s does not increase, (2) the minimum in s does not decrease, and (3) the number of children who hold the minimum number of candies strictly decreases. These properties ensure that repeated application of $circNext$ leads to a situation where the maximum and the minimum in s are the same. More formally, for any list s of even numbers, the following theorem holds:

$$\exists n.\ \max(circNext^{(n)}(s)) = \min(circNext^{(n)}(s))$$

where $f^{(0)}(x) = x$ and $f^{(n+1)}(x) = f(f^{(n)}(x))$. Consequently, eventually all the children will hold the same amount of candy. Following this proof strategy, we established this theorem in Isabelle.

```
 1  datatype Event = left "nat" | right "nat" | mid "nat"
 2  datatype PN = Child "nat" | ChildL "nat*nat" | ChildR "nat"
 3
 4  consts PNdef :: "PN ⇒ (PN, Event) proc"
 5  recdef PNdef "{}"
 6   "PNdef (Child(n))
 7     = (left ! (n div 2) -> $ChildR(n div 2)) [+]
 8       (right ? x -> $ChildL(n,x))"
 9   "PNdef (ChildL(n,x))
10     = left ! (n div 2) -> $Child(fill(n div 2 + x))"
11   "PNdef (ChildR(n))
12     = right ? x -> $Child(fill(n + x))"
13  defs (overloaded) Set_PNfun_def [simp]: "PNfun == PNdef"
14
15  consts
16   CircCh :: "nat list ⇒ (PN, Event) proc"
17   LineCh :: "nat list ⇒ (PN, Event) proc"
18
19  recdef LineCh  "measure(λs. length(s))"
20   "LineCh([n]) = $Child(n)"
21   "LineCh(n#s) = $Child(n) <---> LineCh(s)"
22
23  recdef CircCh  "{}"
24   "CircCh(n#s) = $Child(n) <===> LineCh(s)"
```

Figure 4. The encoded process `CircCh` into CSP-Prover

1		$\texttt{CircCh}(\langle 6,2,4,8\rangle)$	
2	$=$	$\texttt{Child}(6) \Longleftrightarrow (\texttt{Child}(2) \longleftrightarrow \texttt{Child}(4) \longleftrightarrow \texttt{Child}(8))$	
3	$\rightarrow$	$\texttt{ChildL}(6,1) \Longleftrightarrow (\texttt{ChildR}(1) \longleftrightarrow \texttt{Child}(4) \longleftrightarrow \texttt{Child}(8))$	by 2nd Child's pass
4	$\rightarrow$	$\texttt{ChildL}(6,1) \Longleftrightarrow (\texttt{ChildR}(1) \longleftrightarrow \texttt{ChildL}(4,4) \longleftrightarrow \texttt{ChildR}(4))$	by 4th Child's pass
5	$\rightarrow$	$\texttt{ChildL}(6,1) \Longleftrightarrow (\texttt{Child}(4) \longleftrightarrow \texttt{Child}(6) \longleftrightarrow \texttt{ChildR}(4))$	by 3rd Child's pass
6	$\rightarrow$	$\texttt{ChildL}(6,1) \Longleftrightarrow (\texttt{ChildL}(4,3) \longleftrightarrow \texttt{ChildR}(3) \longleftrightarrow \texttt{ChildR}(4))$	by 3rd Child's pass

Figure 5. An example of consecutive internal transitions from $\texttt{CircCh}(\langle 6,2,4,8\rangle)$

3.2 An asynchronous version

The proof of Section 3.1 presumes that the children can always pass on candies, e.g. there is no deadlock possible. Further on, all children are globally synchronized, i.e. all of them pass half of candies to the left in one step, probably the teacher controls the passing.

In the process $\texttt{CircCh}(s)$, however, each child can pass half her/his candies whenever her/his left child can receive them. In this case, the time at which a child passed her/his candies can be different from the time at which another child does so. Take for example the process $\texttt{CircCh}(\langle 6,2,4,8\rangle)$. This process can perform the consecutive transitions shown in Fig. 5. From the first state in line 1 to the last state in line 6, the third child passes half of her/his candies twice, while the first child does not pass her/his candies. In the process $\texttt{CircCh}(s)$, since only the pass by the first child is observable, all transitions in Fig. 5 are not observed (i.e. they are internal). Therefore, the solution given in Section 3.1 is not sufficient for the asynchronous passing in $\texttt{CircCh}(s)$. To deal with $\texttt{CircCh}(s)$, we need a framework for analyzing such behaviour in concurrent processes.

3.3 Proofs in CSP

CSP provides a number of *models* to be selected according to the verification purpose. For example, *failures-equivalence* $=_{\mathcal{F}}$ based on the stable-failures model, which is one of main CSP models, can distinguish between deterministic choice and non-deterministic choice. For example,

$$(a \rightarrow b \rightarrow P_1) \Box (a \rightarrow c \rightarrow P_2) \neq_{\mathcal{F}} a \rightarrow ((b \rightarrow P_1) \Box (c \rightarrow P_2))$$

where the environment can select b or c after a in the right hand side, while it cannot select in the left hand side. Note that they are equal in the traces model.

In addition, *failures-refinement* $\sqsubseteq_{\mathcal{F}}$ is suitable for detecting deadlock and analyzing safety properties. It is also available for analyzing liveness properties if processes are livelock-free. Intuitively, if Q refines P, written $P \sqsubseteq_{\mathcal{F}} Q$, then Q is obtained from P by pruning non-deterministic choices. For example,

$$\begin{array}{l}(a \rightarrow b \rightarrow P_1) \Box (a \rightarrow c \rightarrow P_2) \ \sqsubseteq_{\mathcal{F}} \ a \rightarrow b \rightarrow P_1 \\ \quad a \rightarrow ((b \rightarrow P_1) \Box (c \rightarrow P_2)) \ \not\sqsubseteq_{\mathcal{F}} \ a \rightarrow b \rightarrow P_1\end{array}$$

In CSP, nondeterminism can be expressed by the internal choice operator $\sqcap$ more clearly. Intuitively, $P \sqcap Q$ behaves P or Q, but the choice cannot be controlled from other process. Therefore, for example,

$$\begin{array}{l}(a \rightarrow b \rightarrow P_1) \Box (a \rightarrow c \rightarrow P_2) \\ \quad =_{\mathcal{F}} \ \ a \rightarrow ((b \rightarrow P_1) \sqcap (c \rightarrow P_2))\end{array}$$

where note that the environment cannot select b or c after a in the right hand side either because the selection is internally decided. By the internal choice, loose processes can be expressed. For example, the following process $A(n)$ requires that inc or dec can be iteratively performed.

$$A(n) \ = \ (inc!n \rightarrow A(n+1)) \sqcap (dec!n \rightarrow A(n-1))$$

Therefore, for example, all the following three processes refine $A(n)$:

$$\begin{array}{l}C_1(n) \ = \ inc!n \rightarrow C_1(n+1) \\ C_2(n) \ = \ inc!n \rightarrow dec!(n+1) \rightarrow C_2(n) \\ C_3(n) \ = \ (inc!n \rightarrow C_3(n+1)) \Box (dec!n \rightarrow C_3(n-1))\end{array}$$

CSP provides a set of rewriting rules (CSP-*laws*) for proving refinement relation between processes. Over the model $\mathcal{F}$, the refinement relation can be proven by syntactically rewriting process expressions [9]. For example, the following equalities can be proven by the step-laws and the commutative law for the parallel operator.

$$\begin{array}{lll} & (a \rightarrow P_1) \ [\![a]\!] \ (b \rightarrow a \rightarrow P_2) & \\ =_{\mathcal{F}} & b \rightarrow ((a \rightarrow P_1) \ [\![a]\!] \ (a \rightarrow P_2)) & \text{by (step)} \\ =_{\mathcal{F}} & b \rightarrow a \rightarrow (P_1 \ [\![a]\!] \ P_2) & \text{by (step)} \\ =_{\mathcal{F}} & b \rightarrow a \rightarrow (P_2 \ [\![a]\!] \ P_1) & \text{by (commute)}\end{array}$$

Especially, the step laws are important for sequentializing concurrent processes.

The other important proof technique is fixed point induction which is useful for analyzing infinite-state processes. Intuitively, it is induction on behaviour. For the infinite-processes $A(n)$ and $C_1(n)$ used above, fixed point induction allows us to prove $A(n) \sqsubseteq_{\mathcal{F}} C_1(n)$ for any n by assuming that this refinement holds after one cycle. Thus, the refinement relation can be proven as follows:

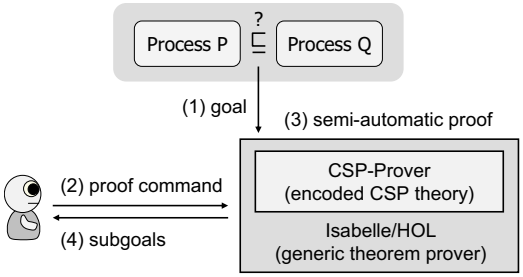

Figure 6. Interactive proof of refinement relation

$$\begin{array}{lll} A(n) & =_{\mathcal{F}} & (inc!n \rightarrow A(n+1)) \sqcap (dec!n \rightarrow A(n-1)) \\ & \sqsubseteq_{\mathcal{F}} & (inc!n \rightarrow A(n+1)) \qquad \text{by refinement} \\ & \sqsubseteq_{\mathcal{F}} & (inc!n \rightarrow C_1(n+1)) \qquad \text{by induction} \\ & =_{\mathcal{F}} & C_1(n)\end{array}$$

3.4 Proving with CSP-Prover

Our tool CSP-Prover [7, 8, 9, 10, 11] is based on is an interactive theorem prover Isabelle [14], which allows one to prove new theorems by semi-automatically applying *rules* which are pre-proven theorems. Successfully proved theorems can be stored and used later as new rules. Therefore, the proof-ability of Isabelle can be extended by adding new definitions and proving new theorems.

CSP-Prover contains fundamental theorems such as fixed point theorems, the definitions of CSP syntax and semantics, many CSP-laws, and also semi-automatic proof tactics for the verification of refinement relations. Fig. 6 shows the interactive proof procedure for a refinement relation: first, (1), the refinement statement is entered into CSP-Prover as a so-called (proof)*goal*; then, (2), the user enters a *proof command*; a command controls the way, (3), in which Isabelle tries to prove the goal by applying CSP theory as automatically as possible; finally, (4), the results of this proof process are displayed as subgoals. Should there be open subgoals left, then the proof is not completed yet, and Isabelle awaits further commands. A proof is successfully finished when there is no open subgoal left. One advantage of this approach is that it can quite elegantly deal with infinite structures, for instance, by using induction. This enables CSP-Prover to verify also infinite state systems [8]. Thanks to the deep encoding, CSP-Prover also can be used to establish new theorems on CSP [9].

3.5 Proof of the puzzle in CSP

Now we return to the Uniform Candy Distribution Puzzle. As explained in Section 3.1, the next numbers of candies can be estimated by the function $circNext(s)$. Thus,

we expect the concurrent process $\texttt{CircCh}(s)$ to be the refinement of a sequential process CircSq for any list s such that $length(s) \geq 2$, therefore

$$\texttt{CircSq}(s) \sqsubseteq_{\mathcal{F}} \texttt{CircCh}(s)$$

where we define $\texttt{CircSq}(s)$ as follows:

$$\texttt{CircSq}(s) = left!(hd(s)/2) \rightarrow \texttt{CircSq}(circNext(s))$$

Here, $hd(\langle n \rangle^\frown s) = n$. Provided the above stated refinement relation holds, this means that the number of candies the first child holds eventually converges to some even number[3]. This implies that for each $i \in \{1, \ldots, k\}$, for some even number c_i, the number of candies the i th child holds eventually converges to c_i because we can select any child to be the first one. Here, for any $i < k$, $c_i = c_{i+1}$ can be proven because $c_i = fill(c_i/2 + c_{i+1}/2)$ and $c_k = fill(c_k/2 + c_1/2)$. This means that eventually all the children will hold the same amount of candy.

In fact, we proved the refinement relation $\texttt{CircSq}(s) \sqsubseteq_{\mathcal{F}} \texttt{CircCh}(s)$ for any list s such that the length of s is greater than 1. In the rest of this subsection, we give an outline of our proof.

At first, in order to deal with internal behaviours in $\texttt{CircCh}(s)$ as shown in Fig. 5, the list s of the number of candies is extended with attributes C, L and R, which represents states Child, ChildL, and ChildR, respectively. For example, the two states of lines 3 and 5 in Fig. 5 are expressed by the extended lists $\langle \texttt{L}(6,1), \texttt{R}(1), \texttt{C}(4), \texttt{C}(8) \rangle$ and $\langle \texttt{L}(6,1), \texttt{C}(4), \texttt{C}(6), \texttt{R}(4) \rangle$, respectively. For the rest of this paper, let s range over lists of even numbers and let t range over extended lists with attributes to distinguish them.

Note that the state of the last line in Fig. 5 cannot perform any internal transition. We call such a state *internally stable*. In our puzzle, an internally stable state can only be of one of the following two forms:

$$\langle \texttt{L}(\cdot,\cdot), \ldots, \texttt{L}(\cdot,\cdot), \texttt{R}(\cdot), \ldots, \texttt{R}(\cdot) \rangle,$$
$$\langle \texttt{L}(\cdot,\cdot), \ldots, \texttt{L}(\cdot,\cdot), \texttt{C}(\cdot), \texttt{R}(\cdot), \ldots, \texttt{R}(\cdot) \rangle$$

On t, we define three recursive functions:

- The function $toStb(t)$ returns the internally stable state of t.
- The function $nextL(t)$ returns the internally stable state after that the first child has sent half of her/his candies to the left at the internally stable t.
- The function $nextR(t, x)$ returns the internally stable state after that the last child has received x candies at the internal stable t.

[3]Note that the numbers of candies in transit states such as ChildL and ChildR are not considered.

We omit here the definitions, however, we illustrate these function by examples (also see Fig. 5):

$$\begin{aligned}
&nextR(nextL(nextL(toStb(\langle \texttt{C}(6), \texttt{C}(2), \texttt{C}(4), \texttt{C}(8) \rangle))), 5) \\
&= nextR(nextL(nextL(\langle \texttt{L}(6,1), \texttt{L}(4,3), \texttt{R}(3), \texttt{R}(4) \rangle)), 5) \\
&= nextR(nextL(\langle \texttt{L}(4,2), \texttt{C}(6), \texttt{R}(3), \texttt{R}(4) \rangle), 5) \\
&= nextR(\langle \texttt{L}(4,3), \texttt{R}(3), \texttt{R}(3), \texttt{R}(4) \rangle, 5) \\
&= \langle \texttt{L}(4,3), \texttt{C}(8), \texttt{R}(4), \texttt{R}(5) \rangle
\end{aligned}$$

With the help of $nextL(t)$ and $nextR(t)$, we define the sequential process $\texttt{LineSq}(t)$ for any internally stable state t as follows:

$$\begin{aligned}
&\texttt{LineSq}(t) \\
&= (\texttt{if}\ guardL(t)\ \texttt{then}\ left!(fst(t)/2) \rightarrow \texttt{LineSq}(nextL(t)) \\
&\quad \texttt{else}\ \ \texttt{STOP}) \Box \\
&\quad (\texttt{if}\ guardR(t)\ \texttt{then}\ right?x \rightarrow \texttt{LineSq}(nextR(t,x)) \\
&\quad \texttt{else}\ \ \texttt{STOP})
\end{aligned}$$

where $guardL(t)$ is true if and only if the attribute of the first child is L or C, $guardR(t)$ is true if and only if the attribute of the last child is R or C, and $fst(t)$ is the number of candies the first child has.

As expected, the following refinement relation can be proven using fixed point induction and CSP-laws:

$$\texttt{LineSq}(toStb(\langle \texttt{C}(n)^\frown t \rangle)) \sqsubseteq_{\mathcal{F}} (\texttt{Child}(n) \longleftrightarrow \texttt{LineSq}(t))$$

for any internally stable t. By induction on the length of t, this implies that

$$\texttt{LineSq}(toStb(\texttt{C}(s)) \sqsubseteq_{\mathcal{F}} \texttt{LineCh}(s)$$

where $\texttt{C}(\langle n_1, \ldots, n_k \rangle) = \langle \texttt{C}(n_1), \ldots, \texttt{C}(n_k) \rangle$ $(k \geq 1)$.

Furthermore, the following refinement relation can be proven using fixed point induction and CSP-laws for any s such that $length(s) \geq 1$:

$$\texttt{CircSq}(\langle n \rangle^\frown s) \sqsubseteq_{\mathcal{F}} (\texttt{Child}(n) \Longleftrightarrow \texttt{LineSq}(toStb(\texttt{C}(s))))$$

Finally, by transitivity of $\sqsubseteq_{\mathcal{F}}$, we have the refinement relation $\texttt{CircSq}(s) \sqsubseteq_{\mathcal{F}} \texttt{CircCh}(s)$ for any s such that $length(s) \geq 2$.

3.6 Proof Support by CSP-Prover

The refinement relation $\texttt{CircSq}(s) \sqsubseteq_{\mathcal{F}} \texttt{CircCh}(s)$ can be proven by the proof strategy explained in Section 3.5. However, the proof is complex, and especially the rewriting by CSP-laws is often tedious and thus error prone. Therefore, we applied CSP-Prover for proving the refinement $\texttt{CircSq}(s) \sqsubseteq_{\mathcal{F}} \texttt{CircCh}(s)$. In fact, CSP-Prover turned out to be extremely helpful to establish this refinement.

Fig. 7 is a typical example of a proof scripts for establishing an equality in CSP-Prover: Line 2 to line 5 state the

```
1 lemma test_two_children_step:
2    "$Child(n)<--->$Child(m) =F
3       (left ! (n div 2) -> ($ChildR(n div 2)<--->$Child(m))[+]
4        right ? x -> ($Child(n)<--->$ChildL(m,x)))
5       [> ($ChildL(n,m div 2)<--->$ChildR(m div 2))"
6 by (auto | tactic{* cspF_hsf_unwind_tac 1 *})+
```

Figure 7. A proof script for proving an equality by CSP-Prover

equality to be proven. In our example, we want to show that two children linearly connected send $(n/2)$ to the left (line 3) or receive a number from the right (line 4) or internally communicate with each other (line 5), where `[>` is the timeout operator $\triangleright$ of CSP. Line 6 is the proof command that actually proves this equality. Here, `auto` is the Isabelle's automatic proof command, which is used for simplifying the data part, and `cspF_hsf_unwind_tac` is a CSP-Prover tactic, which sequentializes concurrent processes by applying various CSP-laws. This theorem can be completely proven by the one line command (line 6). It takes about 10 minutes to prove this equality on a laptop computer (Pentium-M, 1.5GHz). If the user tells in more detail, which CSP-laws shall be applied, the computation time can be shortened.

The overall refinement of Uniform Candy Distribution Puzzle is more complex, but can be proven in a similar way to Fig. 7[4]. Currently, it takes about one hour to prove $\texttt{CircSq}(s) \sqsubseteq_{\mathcal{F}} \texttt{CircCh}(s)$ on a Pentium-M, 1.5GHz.

3.7 Alternative proof tools

[18] present a theorem prover for CSP which is based on the theorem prover PVS [15].

It is possible to analyze the puzzle for single instances of fixed size and with a fixed initial distribution of candies with the well-established model checker FDR [12]. FDR checks fully automatically, if a CSP refinement holds. Also the tool HORAE [3], which is based on constraint satisfaction techniques, can deal with such single instances. Furthermore, it should be possible to fully automatically analyze such single instances using other model checkers like SPIN [6] or SMV [13]. These tools check if a system *satisfies* a property like deadlock-freedom, where systems are described as finite state machines and properties are formulated in temporal logic. Fig. 8 summarizes these difference between CSP-Prover and model checkers. Note that thanks to the CSP modelling approach of expressing properties and systems in the same language, *stepwise* refinement is available in a natural way.

[4]The tactic `cspF_hsf_unwind_tac` is available for both of the refinement $\sqsubseteq_{\mathcal{F}}$ and the equality $=_{\mathcal{F}}$.

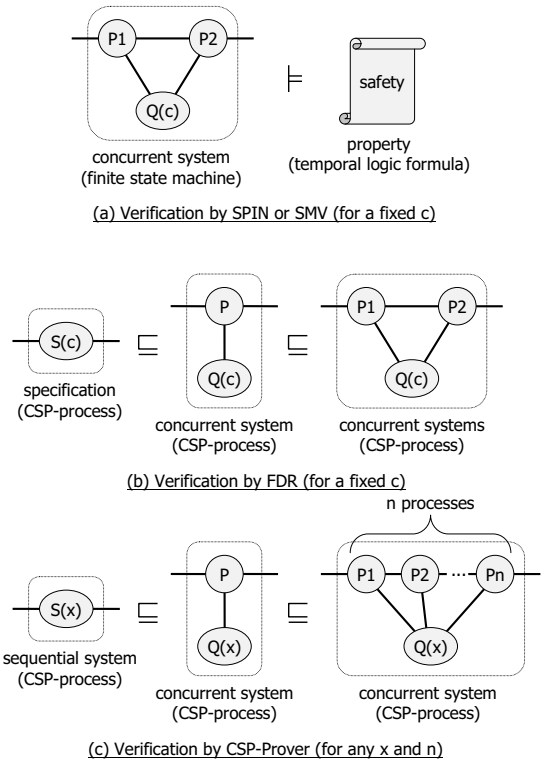

Figure 8. Comparison of verification style

4 Conclusion

By solving the Uniform Candy Distribution Puzzle we have demonstrated that CSP-Prover can deal with scalable distributed systems. Depending on the requirements, such system descriptions can easily be adapted to a different number of processes involved, also their initial configuration can be adapted. In our example, the stabilization theorem is available for any number of children and for any initial candy distribution among the children.

As yet another example of a scalable distributed system, we have analyzed a systolic array in [11]. More details about CSP-Prover can be found in the User Guild for CSP-Prover download-able from CSP-Prover's web-site [7]. CSP-Prover is under continuous development, where we

concentrate on techniques that allow for a higher degree in proof automation.

Acknowledgement

We would like to thank Faron G Moller for bringing the Uniform Candy Distribution Puzzle to our attention, and Erwin R Catesbeiana (Jr) to be such an entertainment for the children. This work was supported by EPSRC as Project EP/D037212/1 and by KAKENHI under reference 20500023.

References

[1] A. Abdallah, C. Jones, and J. Sanders, editors. *Communicating Sequential Processes, the first 25 years*, LNCS 3525. Springer, 2005.

[2] T. Bohman, O. Pikhurko, A. Frieze, and D. Sleator. Puzzle 6: Uniform candy distribution. `http://www.cs.cmu.edu/puzzle/puzzle6.html`.

[3] J. S. Dong, P. Hao, J. Sun, and X. Zhang. A reasoning method for timed CSP based on constraint solving. In *ICFEM 2006*, pages 342–359, 2006.

[4] F. Dressler. *Self-Organization in Sensor and Actor Networks*. Wiley, 2007.

[5] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[6] G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2004.

[7] Y. Isobe and M. Roggenbach. Webpage on CSP-Prover. `http://staff.aist.go.jp/y-isobe/CSP-Prover/CSP-Prover.html`.

[8] Y. Isobe and M. Roggenbach. A generic theorem prover of CSP refinement. In *TACAS 2005*, LNCS 3440. Springer, 2005.

[9] Y. Isobe and M. Roggenbach. A complete axiomatic semantics for the CSP stable failures model. In *CONCUR 2006*, LNCS. Springer, 2006.

[10] Y. Isobe and M. Roggenbach. Proof principles for process algebra - CSP-Prover in practice. In *LDIC 2007*, LNCS. Springer, 2008 (to be published).

[11] Y. Isobe, M. Roggenbach, and S. Gruner. Extending CSP-Prover by deadlock-analysis: Towards the verification of systolic arrays. In *FOSE 2005*, Japanese Lecture Notes Series 31. Kindai-kagaku-sha, 2005.

[12] F. S. E. Limited. Failures-divergence refinement: FDR2. `http://www.fsel.com/`.

[13] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[14] T. Nipkow, L. C. Paulon, and M. Wenzel. *Isabelle/HOL*. LNCS 2283. Springer, 2002.

[15] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *CADE 92*, Lecture Notes in Artificial Intelligence. Springer, 1992.

[16] A. Roscoe. *The theory and practice of concurrency*. Prentice Hall, 1998.

[17] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. *The Modelling and Analysis of Security Protocols: the CSP Approach*. Addison-Wesley, 2001.

[18] K. Wei and J. Heather. Embedding the stable failures model of CSP in PVS. In *IFM'05*, LCNS 3771. Springer, 2005.