ソフトウェア紹介

CSP-Prover – a Proof Tool for the Verification of Scalable Concurrent Systems

Yoshinao Isobe　　Markus Roggenbach

1 Introduction

The *process algebra* CSP [1] [4] [15] [16] is a formal method devoted to the modelling as well as to the analysis and verification of concurrent systems. It is a speciality of CSP that it captures both, the concurrent system as well as its desired properties, as specifications: Let *Sys* be the formal CSP model of a concurrent system, let P be a property formulated in CSP – such a property could, for instance, be deadlock-freedom. In such a setting, the statement $P \sqsubseteq Sys$, read '*Sys* is a refinement of P', expresses that the property P holds for the concurrent system *Sys*. In the proof of such a statement the process algebraic laws of CSP play a vital role: Thanks to completeness results, see e.g. [8] [15], most refinement statements can be proven by solely applying process algebraic laws.

Isabelle [13] is an interactive theorem prover that allows one to prove new theorems by semi-automatically applying *rules* which are pre-proven theorems. Then, 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.

Our tool CSP-Prover [6] [7] [8] [9] [10] provides a deep encoding of CSP in Isabelle. 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. 1 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*; 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 [7]. Thanks to the deep encoding, CSP-Prover also can be used to establish new theorems on CSP [8].

In this paper we introduce CSP-Prover by demonstrating how to model and analyze the famous Uniform Candy Distribution Puzzle†1. The puzzle ex-

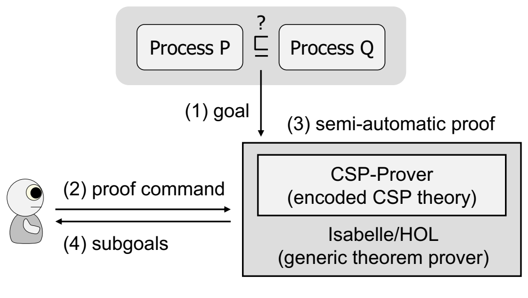


Fig. 1 Interactive proof of refinement relation

†1 It appears to be impossible to name the inventor

CSP-Prover —スケーラブルな並行システムの検証のための証明器

磯部祥尚, 産業技術総合研究所, National Institute of Advanced Industrial Science and Technology (Japan).
マーカス・ローゲンバッハ, スウォンジー大学, Swansea University (UK).

コンピュータソフトウェア, Vol.25, No.4 (2008), pp.85–92.
2008 年 1 月 18 日受付.

hibits many properties typical of scalable concurrent systems: (scalability) the puzzle's size is given as a parameter, i.e. actually a class of systems shall be analyzed; (parameterization) the interchange of candies between the children depends on the initial distribution of candies, i.e. again a class of systems shall be analyzed; (local activity) the children act independently of each other, i.e. there is no global clock; (global result) the children's interaction finally leads to a stable situation.

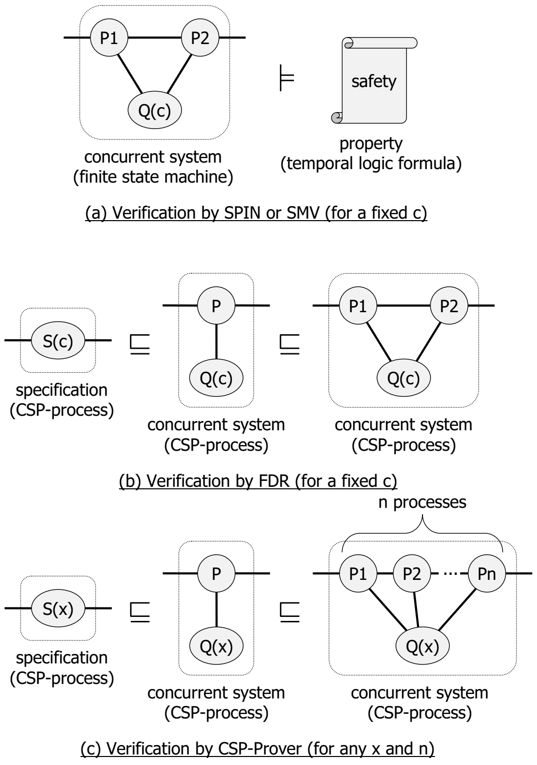
It is possible to analyze the puzzle for single instances of fixed size and with a fixed initial distribution of candies by the well-established model checker FDR [11] for Csp. FDR checks fully automatically, if a Csp refinement holds. Also the tool HORAE [3], which is based on constraint satisfaction 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 [5] or SMV [12]. 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. 2 summarizes these differences 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 in tools for Csp, e.g. FDR and Csp-Prover.

Proofs in Csp-Prover are semi-automatic only, however, Csp-Prover comes with the benefit that *scalable* concurrent systems can be analyzed for any initial values and any number of processes. In [17] a similar proof tool for Csp is presented based on the theorem prover PVS [14].

The paper is organised as follows: First, we introduce the Uniform Candy Distribution Puzzle, model it as a concurrent process in Csp, and encode it within the input language of Csp-Prover. Then, we 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

We explain how to model concurrent systems in CSP and how to encode them in Csp-Prover by means of the Uniform Candy Distribution Puzzle.



P1
P2
Q(c)
⊨
safety
concurrent system (finite state machine)
property (temporal logic formula)
(a) Verification by SPIN or SMV (for a fixed c)
S(c)
⊑
P
Q(c)
⊑
P1
P2
Q(c)
specification (CSP-process)
concurrent system (CSP-process)
concurrent system (CSP-process)
(b) Verification by FDR (for a fixed c)
n processes
S(x)
⊑
P
Q(x)
⊑
P1
P2
Pn
Q(x)
specification (CSP-process)
concurrent system (CSP-process)
concurrent system (CSP-process)
(c) Verification by CSP-Prover (for any x and n)

Fig. 2 Comparison of verification style

2.1 Introduction to the puzzle

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

Uniform Candy Distribution Puzzle: k children are sitting in a circle (see Fig. 3). 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.

Prove: Eventually all children will have the same amount of candy.

The behaviour of each child is expressed by the transition graph shown in Fig. 4. It consists of three parameterized states, `Child`(n), `ChildL`(n, x), and `ChildR`(n). Here, `Child`(n) is the initial state, n is the number of candies a child holds, and the func-

of the puzzle, one reference, however, is [2].

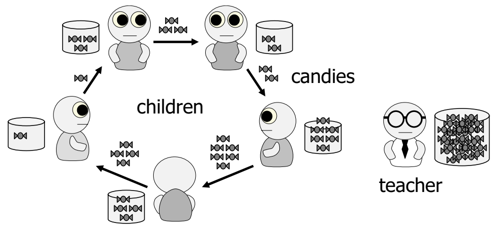


Fig. 3 Uniform Candy Distribution Puzzle

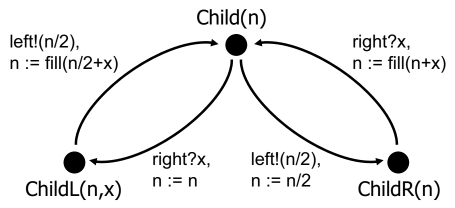


Fig. 4 Transition graph of each child

tion *fill* is defined as

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

The event $left!(n/2)$ means to send the value $(n/2)$ to the left, the event $right?x$ means to receive a number from the right and to bind the variable x to the received number. Therefore, each child sends half of her/his candies to the left child and receives x candies from the right child. Sending and Receiving can happen in either order. 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.

2.2 Modelling the puzzle in CSP

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

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

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. In our example of CSP processes, the process definition of `Child`, `ChildL`, and `ChildR` consists essentially of infinitely many equations, since the parameter n is an arbitrary even number. 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 the 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]:

$$P \longleftrightarrow Q = (P[\![right \leftrightarrow mid]\!]\ |[\,mid\,]|\ Q[\![left \leftrightarrow mid]\!]) \backslash mid$$
$$P \Longleftrightarrow Q = (P\ |[\,left, right\,]|\ Q[\![left \leftrightarrow right]\!]) \backslash right$$

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

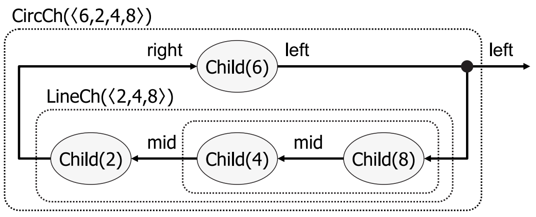


Fig. 5 Structure of $\texttt{CircCh}(\langle 6, 2, 4, 8\rangle)$

†2 For simplicity, we use in this paper typical Csp convention. For example, more precisely the operator $|[mid]|$ represents $|[\{mid(n) \mid n \in Nat\}]|$.

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

Fig. 6 The encoded process `CircCh` into CSP-Prover

obtained by renaming both of *right* in P and *left* in Q to a new common name *mid*, composing them via *mid*, and hiding *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. 5 shows the structure of the process `CircCh`$(\langle 6, 2, 4, 8 \rangle)$.

2.3 Encoding the puzzle in CSP-Prover

It is easy to encode CSP-processes into CSP-Prover. Fig. 6 shows the encoded process `CircCh`, where some definitions such as `<--->` have been omitted. The complete code is available at CSP-Prover's web-site [6].

In Fig. 6, 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 parameterized type `proc` is provided by CSP-Prover. Next, lines 4–9 define the function `PNdef` which maps process names of the type `PN` to processes of the 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 □ 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 6–9. When defining a recursive function by `recdef`, Isabelle requires a measure to guarantee termination of the defined function. For example, line 16 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 10 declares the function `PNdef` as the function `PNfun` which is 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 16–21.

3 Verification

We show how to verify concurrent systems in CSP with the help of CSP-Prover by using the example of the Uniform Candy Distribution Puzzle.

3.1 A known solution

Solutions of the Uniform Candy Distribution Puzzle have already been given, e.g. in the webpage [2]. One solution is as follows: Let s be the list of

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

Fig. 7 An example of consecutive internal transitions from $\mathtt{CircCh}(\langle 6,2,4,8\rangle)$

even numbers of candies 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):

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

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

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

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:

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

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.

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 $\mathtt{CircCh}(s)$, however, each child can pass half of 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 $\mathtt{CircCh}(\langle 6,2,4,8\rangle)$. This process can perform the consecutive transitions shown in Fig. 7. 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 $\mathtt{CircCh}(s)$, since only the pass by the first child is observable, all transitions in Fig. 7 are not observed (i.e. they are internal). Therefore, the solution given in Section 3. 1 is not sufficient for the asynchronous passing in $\mathtt{CircCh}(s)$. To deal with $\mathtt{CircCh}(s)$, we need a framework for analyzing such behaviour in concurrent processes.

3. 3 Proofs in CSP

Csp is a formal method for modelling and analyzing behaviour of concurrent processes. 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 $\mathcal{F}$, which is one of main Csp models, can distinguish between deterministic choice and non-deterministic choice. For example,

$$\begin{aligned}
&(a \to b \to P_1) \,\Box\, (a \to c \to P_2) \\
&\neq_{\mathcal{F}}\ \ a \to ((b \to P_1) \,\Box\, (c \to P_2))
\end{aligned}$$

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,

$$(a \to b \to P_1) \,\Box\, (a \to c \to P_2) \;\sqsubseteq_{\mathcal{F}}\; a \to b \to P_1$$
$$a \to ((b \to P_1) \,\Box\, (c \to P_2)) \;\not\sqsubseteq_{\mathcal{F}}\; a \to b \to P_1$$

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,

$$\begin{aligned}&(a \to b \to P_1) \,\Box\, (a \to c \to P_2)\\ =_{\mathcal{F}}\;& a \to ((b \to P_1) \sqcap (c \to P_2))\end{aligned}$$

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.

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

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

$$\begin{aligned}C_1(n) &= inc!n \to C_1(n+1)\\ C_2(n) &= inc!n \to dec!(n+1) \to C_2(n)\\ C_3(n) &= (inc!n \to C_3(n+1)) \,\Box\, (dec!n \to C_3(n-1))\end{aligned}$$

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 [8]. For example, the following equalities can be proven by the step-laws and the commutative law for the parallel operator.

$$\begin{aligned}&(a \to P_1) \,[\![a]\!]\, (b \to a \to P_2) &\\ =_{\mathcal{F}}\;& b \to ((a \to P_1) \,[\![a]\!]\, (a \to P_2)) &\text{by (step)}\\ =_{\mathcal{F}}\;& b \to a \to (P_1 \,[\![a]\!]\, P_2) &\text{by (step)}\\ =_{\mathcal{F}}\;& b \to a \to (P_2 \,[\![a]\!]\, P_1) &\text{by (commute)}\end{aligned}$$

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:

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

3.4 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 `CircCh`(s) to be the refinement of a sequential process `CircSq` for any list s of even numbers such that *length*$(s) \geq 2$, therefore

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

where we define `CircSq`(s) as follows:

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

Here, $\mathit{hd}(\langle n \rangle \frown s) = n$. 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 $c_i = \mathit{fill}(c_i/2 + c_{i+1}/2)$ and $c_k = \mathit{fill}(c_k/2 + c_1/2)$. It means that eventually all the children will hold the same amount of candy.

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

At first, in order to deal with internal behaviours in `CircCh`(s) as shown in Fig. 7, 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. 7 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. 7 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 each extended list t, we define three recursive

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

```
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 *})+
```

Fig. 8 A proof script for proving an equality by CSP-Prover

functions:

- The function $\mathit{toStb}(t)$ returns the internally stable state of t.
- The function $\mathit{nextL}(t)$ 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 $\mathit{nextR}(t, x)$ returns the internally stable state after that the last child has received x candies at the internally stable t.

We omit here the definitions, however, we illustrate these function by examples (also see Fig. 7):

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

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

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

where $\mathit{guardL}(t)$ is true if and only if the attribute of the first child is $\mathtt{L}$ or $\mathtt{C}$, $\mathit{guardR}(t)$ is true if and only if the attribute of the last child is $\mathtt{R}$ or $\mathtt{C}$, and $\mathit{fst}(t)$ 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:

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

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

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

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

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

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

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

3.5 Proof Support by CSP-Prover

By using CSP-Prover, we could prove that a certain set of CSP-laws is sound and complete for stable failures equivalence [8]. This result means that CSP-Prover fully supports the proof of the refinement and the equality by CSP-laws, where the correctness of the proof is guaranteed by Isabelle. Furthermore, we have also implemented tactics in CSP-Prover for applying suitable CSP-laws as automatically as possible.

The refinement relation $\mathtt{CircSq}(s) \sqsubseteq_{\mathcal{F}} \mathtt{CircCh}(s)$ can be proven by the proof strategy explained in Section 3.4. 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 $\mathtt{CircSq}(s) \sqsubseteq_{\mathcal{F}} \mathtt{CircCh}(s)$. In fact, CSP-Prover turned out to be extremely helpful to establish this refinement.

Fig. 8 is a typical example of a proof scripts for establishing an equality in CSP-Prover: Line 2 to line 5 state the 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. 8[†4]. And, the lemmas about lists, natural numbers, and so on can be proven by Isabelle's theories. Currently, it takes about one hour to prove $\texttt{CircSq}(s) \sqsubseteq_{\mathcal{F}} \texttt{CircCh}(s)$ on a laptop computer (Pentium-M, 1.5GHz).

4 Conclusion

In order to solve the Uniform Candy Distribution Puzzle, both data and processes have to be analyzed at the same time. In other words, we cannot analyze it by separating processes and data or by replacing values with abstract symbols.

By solving the puzzle we have demonstrated that Csp-Prover can deal with scalable concurrent systems, which can easily change the number of processes and ranges of data depending on requirements, for example, this result is available for any number of children and for any initial numbers of candies children have. As the other example of scalable concurrent systems, a systolic array is analyzed in [10].

More details about Csp-Prover can be found in the User Guild for Csp-Prover downloadable from Csp-Prover's website [6]. 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 Professor 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 the EPSRC Project EP/D037212/1 and by KAKENHI under reference 20500023.

References

[1] Abdallah, A., Jones, C. and Sanders, J.(eds.): *Communicating Sequential Processes, the first 25 years*, LNCS 3525, Springer, 2005.
[2] Bohman, T., Pikhurko, O., Frieze, A. and Sleator, D.: Puzzle 6: Uniform Candy Distribution, http://www.cs.cmu.edu/puzzle/puzzle6.html.
[3] Dong, J. S., Hao, P., Sun, J. and Zhang, X.: A Reasoning Method for Timed CSP Based on Constraint Solving, in *ICFEM 2006*, 2006, pp. 342–359.
[4] Hoare, C. A. R.: *Communicating Sequential Processes*, Prentice Hall, 1985.
[5] Holzmann, G. J.: *The SPIN Model Checker*, Addison-Wesley, 2004.
[6] Isobe, Y. and Roggenbach, M.: Webpage on Csp-Prover, http://staff.aist.go.jp/y-isobe/CSP-Prover/CSP-Prover.html.
[7] Isobe, Y. and Roggenbach, M.: A generic theorem prover of CSP refinement, in *TACAS 2005*, LNCS 3440, Springer, 2005.
[8] Isobe, Y. and Roggenbach, M.: A complete axiomatic semantics for the CSP stable failures model, in *CONCUR 2006*, LNCS 4137, Springer, 2006.
[9] Isobe, Y. and Roggenbach, M.: Proof Principles for Process Algebra - CSP-Prover in Practice, in *LDIC 2007*, Springer, 2008.
[10] Isobe, Y., Roggenbach, M. and Gruner, S.: Extending CSP-Prover by deadlock-analysis: Towards the verification of systolic arrays, in *FOSE 2005*, Japanese Lecture Notes Series 31, Kindai-kagaku-sha, 2005.
[11] Limited, F. S. E.: Failures-divergence refinement: FDR2, http://www.fsel.com/.
[12] McMillan, K. L.: *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.
[13] Nipkow, T., Paulon, L. C. and Wenzel, M.: *Isabelle/HOL*, LNCS 2283, Springer, 2002.
[14] Owre, S., Rushby, J. M. and Shankar, N.: PVS: A Prototype Verification System, in *CADE'92*, Lecture Notes in Artificial Intelligence 607, Springer, 1992.
[15] Roscoe, A.: *The theory and practice of concurrency*, Prentice Hall, 1998.
[16] Ryan, P., Schneider, S., Goldsmith, M., Lowe, G. and Roscoe, B.: *The Modelling and Analysis of Security Protocols: the CSP Approach*, Addison-Wesley, 2001.
[17] Wei, K. and Heather, J.: Embedding the Stable Failures Model of CSP in PVS, in *IFM'05*, LCNS 3771, Springer, 2005.

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