A Tutorial Introduction to Socos

Johannes Eriksson <joheriks@abo.fi>
Åbo Akademi University, Department of IT
Turku, FI-20520, Finland

December 29, 2022

Socos (www.imped.fi/socos) is a free, open-source tool for constructing and verifying invariant-based programs. It consists of an Eclipse-based graphical diagram editor connected to the theorem provers PVS and Yices for discharging verification conditions automatically. This document is a short tutorial introduction to Socos. It guides the reader through the installation of Socos, and shows how to use Socos to verify a simple invariant-based program. It assumes no knowledge of automatic theorem proving, but the reader should be familiar with the basics of invariant-based programming.

1. Introduction

Invariant-based programming (IBP) is a diagrammatic method for developing programs that are correct by construction. The programmer writes the pre- and postconditions and loop invariants as state predicates called *situations* before the actual code. After the situations have been defined, the programmer adds the actual program code—the *transitions*—and proves that each added transition is *consistent* with the invariants. When all transitions have been verified consistent, the programmer checks that the program has no infinite loops—i.e., that it *terminates*—and also that the program reaches the desired postcondition—i.e., that it is *live*.

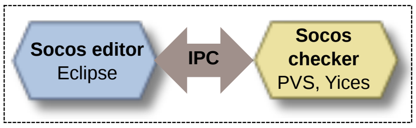
For more information on IBP, see: R.-J. Back. Invariant Based Programming: Basic Approach and Teaching Experience. *Formal Aspects of Computing, 21(3)*, 227–244

Socos is a program construction and verification tool supporting IBP. Invariant diagrams are drawn in a graphical editor in the Eclipse (www.eclipse.org) environment. By the click of a button, Socos sends the verification conditions of the diagram to the automatic theorem prover PVS (pvs.csl.sri.com). The verification conditions are attacked with powerful proof strategies to automatically discharge as many conditions as possible. The default strategy uses the automatic theorem prover Yices (yices.csl.sri.com), which is integrated into PVS, as a catch-all strategy. Only the conditions that were not proved automatically are shown to the user. Specifications and invariants are expressed in the language of PVS. The user can add auxiliary functions, predicates and even entire PVS theories. Socos also includes a small number of domain theories for reasoning about data structures such as arrays.

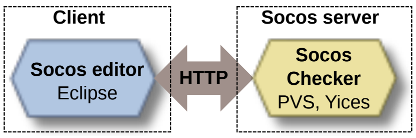
The tutorial is organized as follows. Section 2 describes how to install Socos. Section 3 describes how to use Socos in the context of a simple example program. Section 4 contains contact information for updates and bug reports. Appendix A is a quick-reference to the Socos and PVS language.

2. Installation

Socos is comprised of two components, the *editor* and the *checker*. The editor acts as the front-end to the checker and is distributed as a plug-in to Eclipse. The checker can be set up to run either locally on the same host as the editor, or remotely on another host connected over the network. In the first case the editor communicates with the checker over Unix IPC (pipes), in the second case over HTTP:



Local checking



Remote checking

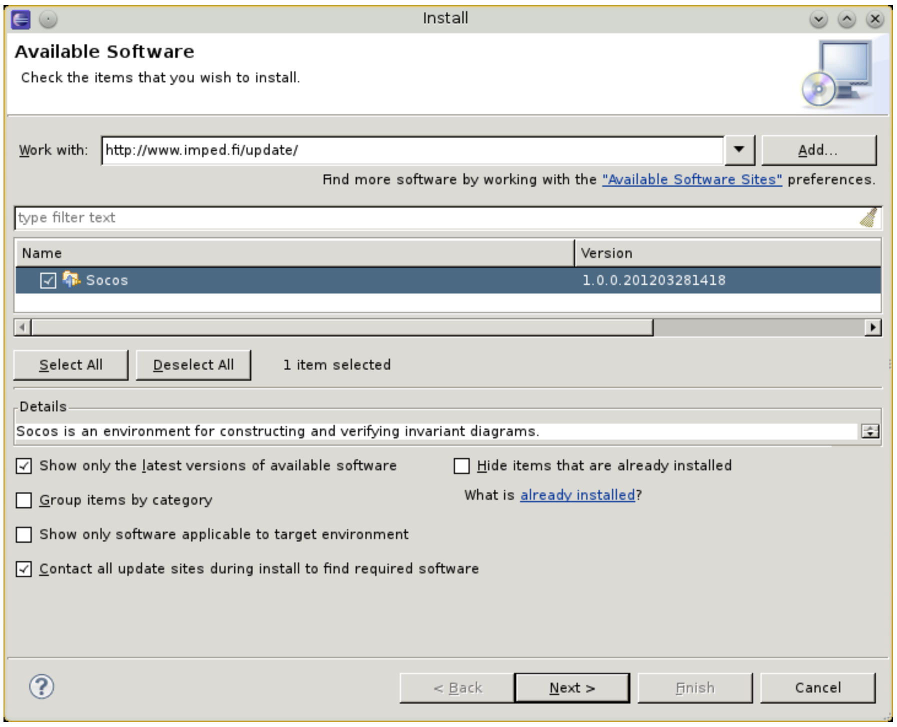
Remote checking allows use of Socos without installing PVS and Yices, only the editor is needed. The remainder of this section describes how to install the diagram editor, as well as setting up remote and local checking. Note that this tutorial does not describe how to set up a checking server.

2.1. Installing the diagram editor

Prerequisites. The diagram editor works on any platform for which Eclipse is available. Eclipse can be downloaded at `www.eclipse.org/downloads`. Version 3.5 (Galileo) or newer is required, though we recommend installing the latest stable version. Various package configurations differing in the set of included plug-ins are available for each Eclipse version. For running Socos, the Eclipse Classic configuration is sufficient.

Installation. Eclipse plug-ins are installed using Eclipse's update manager. To install the Socos editor plug-in, follow the steps below:

1. Launch Eclipse.
2. Select Help ▷ Install new software. A list of the known update sites appears.
3. Add `http://www.imped.fi/update` to the list of update sites, then select Socos from the list of available plug-ins (see figure below). Click next.



Available Software

Check the items that you wish to install.

Work with: http://www.imped.fi/update/

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
☑ Socos	1.0.0.201203281418

1 item selected

Details

Socos is an environment for constructing and verifying invariant diagrams.

☑ Show only the latest versions of available software
☐ Hide items that are already installed
☐ Group items by category
What is already installed?
☐ Show only software applicable to target environment
☑ Contact all update sites during install to find required software

4. Click next again, accepting the license agreement. Socos requires the Graphical Editing Framework (GEF) plug-in, which depending on your configuration of Eclipse may not be pre-installed; the update manager should automatically find and install any missing dependencies.

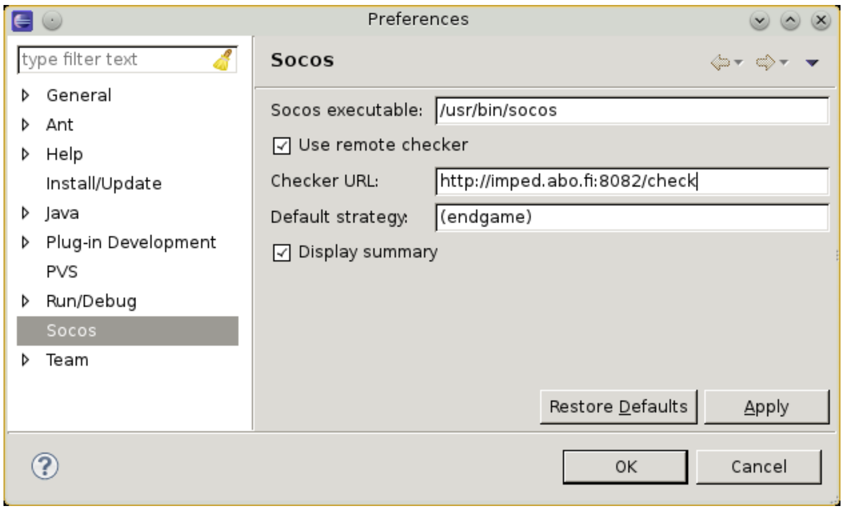
5. Wait for the installation to complete. This process may take several minutes. Restart Eclipse when prompted.

The last step before using Socos is to configure either remote or local checking. This is described in the following sections.

2.2. Configuring remote checking

To configure Socos to use an existing remote checker, open the Eclipse preferences and select the Socos page. Enable Use remote checker and enter the address to a known checking server in the field Checker URL. The field must contain a URL of the form `http://`*hostname*`:`*port*`/check` .

Windows and Linux: Window ▷ Preferences
Mac OS X: Eclipse ▷ Preferences



Socos

Socos executable: /usr/bin/socos

☑ Use remote checker

Checker URL: http://imped.abo.fi:8082/check

Default strategy: (endgame)

☑ Display summary

2.3. Configuring local checking

To set up a local checker, you must first download and install the following packages (unless already installed):

- Python 2, version 2.4 or newer: `www.python.org`
- PVS 5.0, older versions *do not* work: `pvs.csl.sri.com`
- Yices 1.0.X: `yices.csl.sri.com`

Follow the installation instructions in the documentation for package.

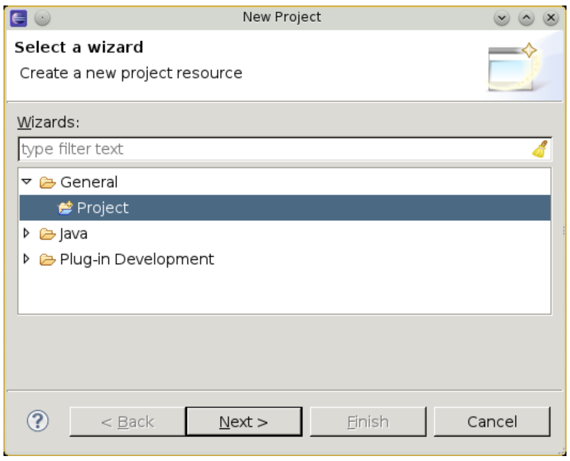
Next, download the Socos checker from `www.imped.fi/socos` . Unpack the Socos package in any location *targetdir*. The checker executable is *targetdir*`/pc/socos` . Socos looks for programs `pvs` and `yices` on the search path (environment variable `PATH`) when executed, so you may need to extend the search path depending on where you installed PVS and Yices. If `pvs` is not found, Socos will not work at all. If `yices` is not found, you will notice it by the default strategy not being able to prove anything. For PVS to run on 64-bit Linux systems it may be necessary to install 32-bit versions of the standard libraries (libc, etc). Once the checker has been installed locally, configure the diagram editor to use it by disabling Use remote checker and ensuring that the Socos executable field contains *targetdir*`/pc/socos` in preferences.

3. Using Socos

In this section, we will construct and verify a simple program in Socos step by step. Before starting this section make sure you have installed the editor and configured either local or remote checking as described in the previous section. If an error occurs, check that recent versions of all the above mentioned packages are installed, and if using remote checking, that the remote host is accessible. If everything seems correctly set up, but the error persists, please file a bug report (see Section 4 for more information).

Start Eclipse and pick a suitable *workspace* location. Once Eclipse is up and running, create a fresh project: New ▷ Project. In the project wizard, select General ▷ Project, then click Next. Type a name for the project, then click Finish.

The *workspace* is the directory in which Eclipse keeps all your projects. Any directory can be used, but concurrently running Eclipse instances must use separate workspaces.


New Project

Select a wizard

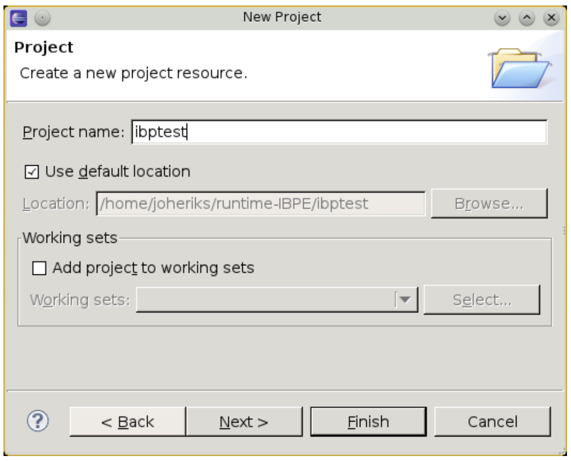
Create a new project resource

Wizards:

type filter text

- General
  - Project
- Java
- Plug-in Development

< Back | Next > | Finish | Cancel


New Project

Project

Create a new project resource.

Project name: ibptest

☑ Use default location

Location: /home/joheriks/runtime-IBPE/ibptest Browse...

Working sets

☐ Add project to working sets

Working sets: Select...

< Back | Next > | Finish | Cancel

Next, we add a Socos file—called a *context*—to the newly created project. Right-click on the project in the *project explorer* and select New ▷ Context in the pop-up menu (if the Context item is not visible in the submenu, select instead Other... ▷ Socos ▷ Context). Give the context any name, then click Finish. The new context will appear as a new tab in the Eclipse workspace.

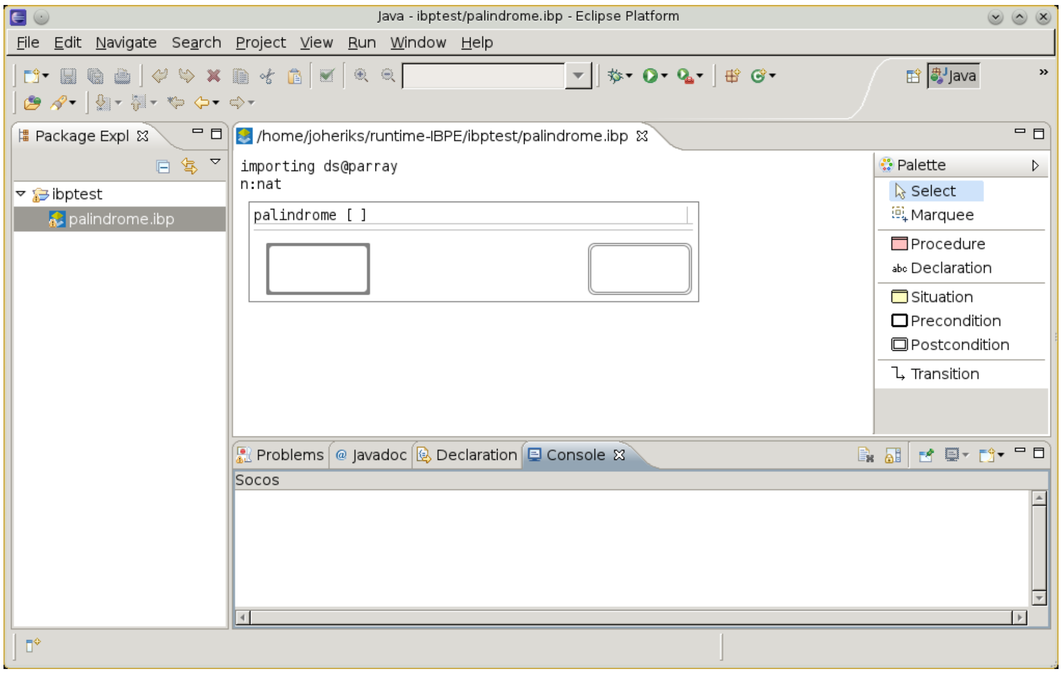
The *project explorer* is the hierarchy browser on the left hand side of the Eclipse workbench. It provides access to all projects in the current Eclipse workspace. If the project explorer is not visible, enable it with Window ▷ Show View ▷ Project Explorer.

A context can contain *declarations* and *procedures* (programs). In this tutorial, we will create a routine that determines if an array of integers is a palindrome—i.e., whether reading the array from the start or from the end results in the same sequence of values. Socos includes a theory for reasoning about arrays, which we must first import into the context. To add a new declaration to the context, activate the New declaration (abc) tool in the tool palette on the right and click on the context area (or right-click on the context area and select Add declaration from the pop-up menu). Click once on the declaration to select it, then click once again to edit it. To move a declaration, drag it to the desired position.

Now add the following to the context (as two separate declarations):

```
importing ds@parray
n:nat
```

The first declaration imports the `ds@parray` theory, which defines a type of arrays with fixed size (`parray`). The second declaration adds a global constant `n` of type `nat` (natural number) to the context. Next, we create a new empty program Activate the New procedure (▭) tool, then click below the added declarations. An empty procedure appears in the document. To change the procedure name, select the procedure and click on the name in the upper left corner of the box. The default name `p1` is not very informative, so change it to something more interesting such as `palindrome`. Your workbench should now be in the following state:



```
importing ds@parray
n:nat
```

```
palindrome [ ]
```

To the right of the procedure name is the (currently empty) parameter list. Below the name follows two compartments holding respectively the local declarations and situations of the procedure. The parameter list and local variable compartments are initially empty, and two empty situations—the pre- and postconditions of the procedure—are added by default to the situation compartment. The precondition (the leftmost situation) is identified by a thick outline, whereas the postcondition (the rightmost situation) has a double outline.

For our procedure to be useful, it must define some input and/or output parameters. Since the procedure is to determine whether an array is a palindrome, it needs an array as input, and an output parameter of Boolean type. Parameters are added similarly to global declarations—by activating the New declaration (abc) tool in the tool palette and clicking on the parameter list (the empty area between the [and] brackets). Next, add the following to the parameter list of `palindrome`:

```
a:parray[n,int]
b:result bool
```

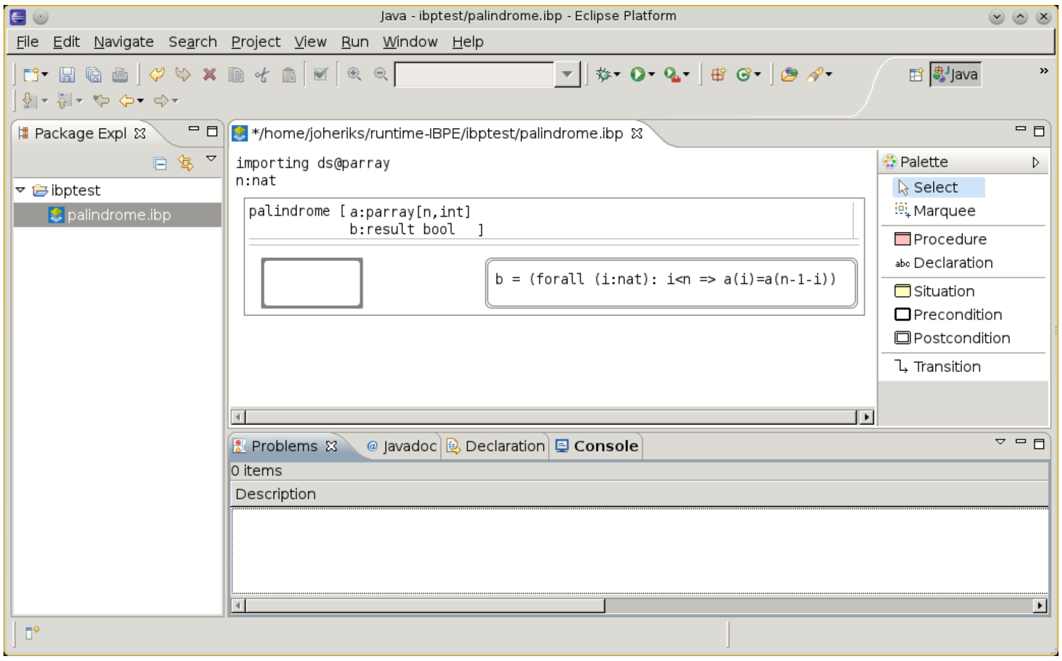
The first line declares a constant parameter called `a` of size `n` and type integer. That `a` is constant means that the program is not allowed to assign to `a`. The second line defines `b` as a *result parameter* of type `bool`, meaning that the program will assign a return value to `b`. However, the program may not make assumptions about the initial value of `b` (if we want to use a parameter for both input and output data, then we should declare it as a `valres` (*value-result*) parameter instead).

The next step is to write the specification of the procedure—namely the pre- and postconditions. Our program does not need additional preconditions—it should accept all arrays, also empty ones. As an empty situation corresponds to `true`, we do not modify the precondition situation. The postcondition, however, should state that `b` is true if and only if `a` is a palindrome. We can consider empty arrays also as palindromes. We state the postcondition in Socos as follows:

```
b = (forall (i:nat):i<n => a(i)=a(n-1-i))
```

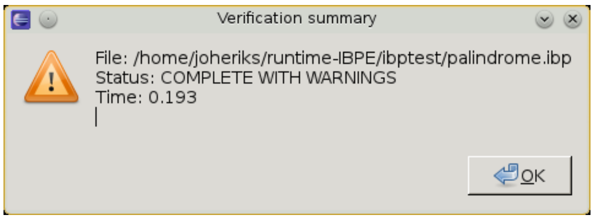
The predicate expresses that `b` has the truth value of the universal quantification on the right hand side. Note that arrays are indexed from 0 to `n` − 1, and that array access is written as `a(i)` for index `i`. This is because `parray`:s are just functions, and array access is function application. PVS is a logic of total functions, meaning that arrays can only be accessed within their domains. PVS checks that array indexes (and function arguments in general) are within the domain of the function (if this check fails, an unsolved TCC (type correctness condition) error will be issued).

Add the above declaration to the postcondition (using the Declaration tool, abc). Your workbench should now look as follows:



```
importing ds@parray
n:nat

palindrome [ a:parray[n,int]
             b:result bool   ]

b = (forall (i:nat): i<n => a(i)=a(n-1-i))
```

Before we start writing the actual program, at this point it is a good idea to ask Socos to check the specification in order to identify potential syntax and type errors. Ensure that the diagram editor is focused, then click the check (✓) toolbar button. The effect of this action is to save the current diagram, then call the Socos checker, and finally display the result of the check. For our specification, Socos reports:

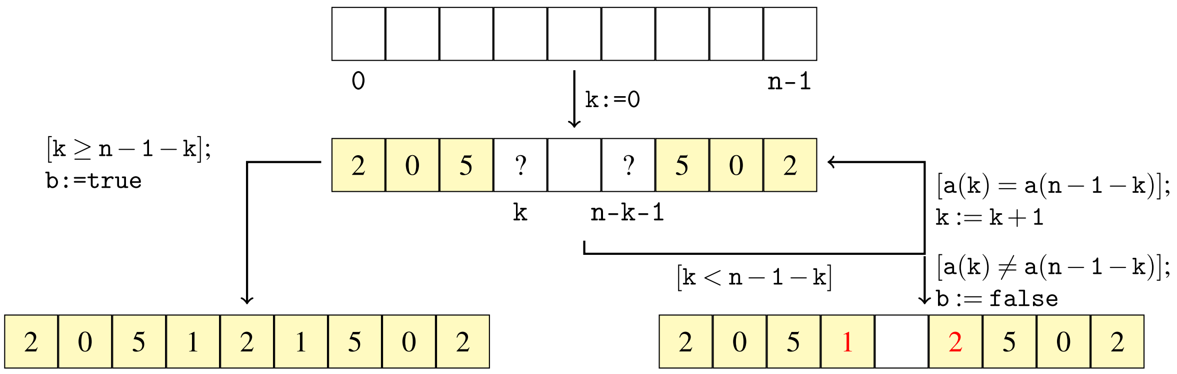


```
File: /home/joheriks/runtime-IBPE/ibptest/palindrome.ibp
Status: COMPLETE WITH WARNINGS
Time: 0.193
```

The exit status COMPLETE WITH WARNINGS means that the checked program had no errors or unproved verification conditions, but that there were some warnings. Other exit statuses are COMPLETE, meaning that there were no errors or warnings, and INCOMPLETE, meaning that the program had errors. The warnings and errors issued by Socos are displayed in the Problems tab, located in the lower part of the workbench. Additionally, elements which have warnings or errors associated with them are marked with squiggly underbars—orange for warnings and red for errors. To quickly view a warning or error in the editor, hover the mouse cursor over the marked element. In this case, Socos warns that the postcondition is unreachable, which is expected since we have not yet added any transitions. Unfinished, but consistent, programs are not considered erroneous by Socos, since it is often useful to check absence of errors (consistency) before all transitions have been added. In the sequel, we will ignore the warnings about liveness and termination until we have added and verified all transitions, after which we will eliminate the warnings by proving liveness and termination.

If the Problems tab is not visible, activate it with the command Window ▷ Show View ▷ Problems.

Now that we are satisfied with the specification, we continue with the implementation. We choose here the most straightforward program for determining whether an array is palindromic: by comparing element by element the first half of the array (in left to right order) to the second half of the array (in right to left order). We start a loop counter k from zero and in each iteration compare the element at index `k` with its counterpart in the second half of the array, i.e., the element at index $\texttt{n}-1-\texttt{k}$. If we find a pair of elements that are different, the array cannot be palindromic and we can immediately return `false`. If the current pair of elements are equal, we advance to the next pair of elements, incrementing `k` by one. When $\texttt{k} \geq \texttt{n}-1-\texttt{k}$ we have compared all element pairs with positive result, and we should hence return `true`. These situations and transitions are depicted in the following figure:



Add the declaration `k:pvar int` to the local declaration compartment of `palindrome`. This introduces `k` as a local variable. From the above figure we read that the invariant maintained by this program is that `k` is in the range $\texttt{0}\ldots\texttt{n}-1-\texttt{k}$, and that all elements to the left of `k` are equal to their counterparts. We can state these predicates as follows in Socos:

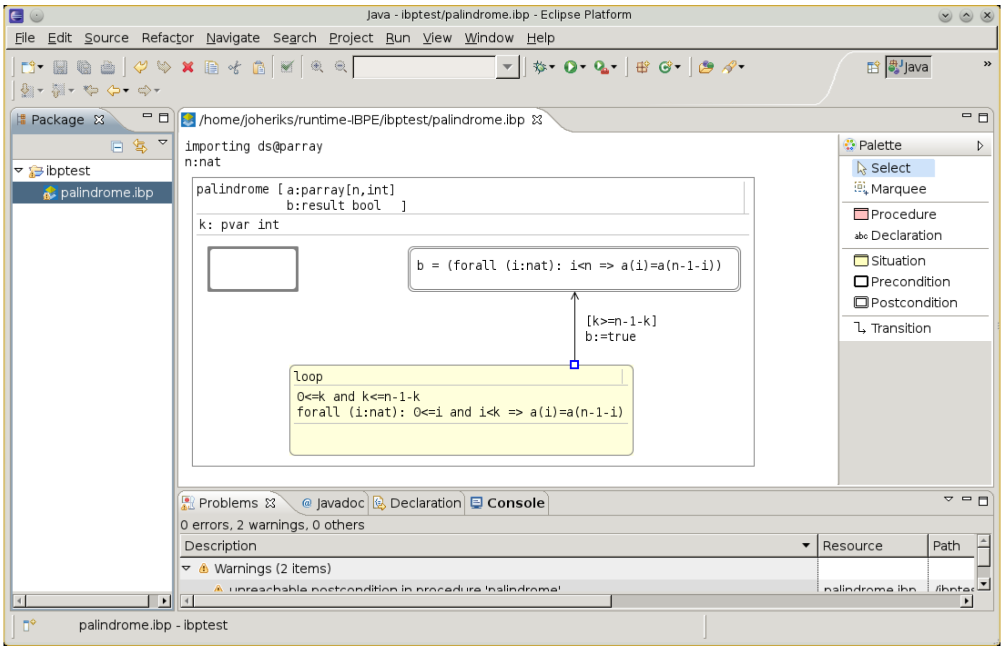
Note: Socos does not support adding variable declarations to situations.

```
0<=k and k<=n-1-k
forall (i:nat):0<=i and i<k => a(i)=a(n-1-i)
```

To add the loop invariants to the diagram, create a new situation (▭) in the procedure, give it a suitable name (e.g., `loop`), and add the above predicates to it. Then check the program again to ensure that the invariants are well defined.

An important aspect of the loop invariant is that it should be sufficiently strong to prove that the exit transition establishes the postcondition. It is therefore usually a good idea to add and check the exit transition first. Activate the transition tool (↳) in the tool palette, then click on the `loop` situation, and finally on the postcondition. This action adds a transition arrow to the diagram. To add a statement to the transition, select the arrow, press Enter, type the statement, and press Enter again. Add the guard `[k>=n-1-k]` followed by the assignment `b:=true`. Add each statement on a separate line (do not separate statements with a semicolon). Your program should now look approximately as follows:

Hint: To add bendpoints to a transition while creating it, hold down shift while clicking. Bendpoints can be added to a selected transition by clicking and dragging the small handles on the center of segments.



```
importing ds@parray
n:nat
palindrome [ a:parray[n,int]
             b:result bool   ]
k: pvar int
b = (forall (i:nat): i<n => a(i)=a(n-1-i))
[k>=n-1-k]
b:=true
loop
0<=k and k<=n-1-k
forall (i:nat): 0<=i and i<k => a(i)=a(n-1-i)
```

```
0 errors, 2 warnings, 0 others
Description                                              Resource        Path
Warnings (2 items)
  unreachable postcondition in procedure 'palindrome'    palindrome.ibp  /ibptes
```

Click the check button. No error is issued, meaning that Socos managed to prove the exit transition consistent. However, Socos issues a warning related to our use of a guard. This is because we have not asked Socos to check liveness yet—the default behavior is to warn that a guard is used, not to try to prove that the situations are live. To turn liveness checking on, right-click on the `loop` situation and select Enable liveness check. Then click the check button again. Now Socos no longer complains about the guard, but instead issues an error for `loop`. Hover the mouse cursor over the red squiggly in the diagram to see the unproved condition:

Liveness checking is set separately for each situation. The current state is indicated by the connection points for outgoing transitions: hollow squares mean that liveness is not checked, whereas solid squares mean that liveness is checked.

procedure 'palindrome', liveness assertion in transition from situation 'loop'
— 0 <= k_0 AND k_0 <= n - 1 - k_0
— FORALL (i: nat): 0 <= i AND i < k_0 => a(i) = a(n - 1 - i)
⊢ (k_0 >= n - 1 - k_0)

The assumptions of the unproved condition are listed before the turnstile (⊢). In this case, the loop invariants are the assumptions. Following the turnstile is the proposition that should be proved—in this case the guard of the exit transition. `k_0` stands for the value of `k` before the transition has started to execute. Each assignment in a transition introduces an incrementally indexed variable: i.e., `k_`*n* represents the value of `k` after the *n*:th assignment. As there is only one outgoing transition, Socos has tried to prove that the guard is true assuming the loop invariant. Since the liveness condition clearly will be false until all transitions originating from `loop` have been added, we postpone liveness checking until the end by right-clicking again on `loop` and selecting Disable liveness check.

Next, we add the initial transition. Draw a transition from the precondition to `loop` with the statement `k:=0`. When we ask Socos to check the program again, it now tells us that the new transition is inconsistent:

procedure 'palindrome', constraint in transition from precondition to situation 'loop'
— k_1 = 0
⊢ 0 <= k_1 AND k_1 <= n - 1 - k_1

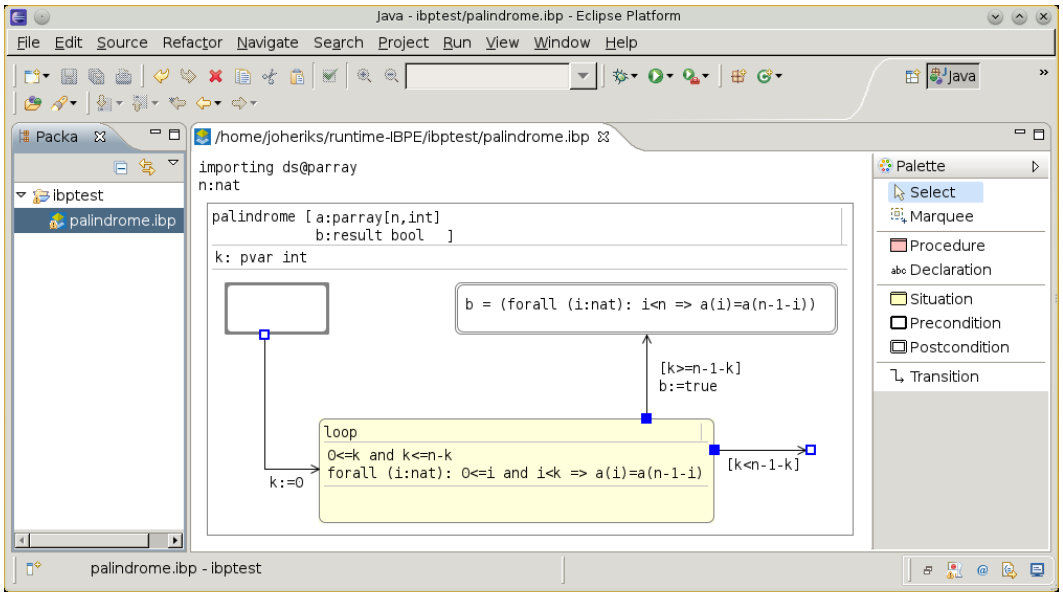
It seems Socos was not able to prove that the first loop invariant is established. We note from the above that after substitution we should prove that `0`≤`n-1-0` (the second conjunct), which is clearly not true for the case `n=0` (but true for `n` > 0). This lets us conclude that our loop invariant is too strong—it cannot be established by the initial transition in all cases! We could make the transition consistent by prefixing it with the guard `[n>0]`, but then we would need to handle the case $\mathtt{n} = 0$ in a separate transition (since our program should work also for empty arrays). Alternatively, we can relax the first loop invariant by extending the upper limit by one element. Taking the latter approach, the new loop invariant becomes:

```
0<=k and k<=n-k
```

After this change, both transitions are proved automatically.

For the case $\mathtt{k} < \mathtt{n} - 1 - \mathtt{k}$ we have two branches: one transition back to `loop` incrementing `k` when $\mathtt{a}(\mathtt{k}) = \mathtt{a}(\mathtt{n} - 1 - \mathtt{k})$, and one transition to the postcondition returning `false` when $\mathtt{a}(\mathtt{k}) \neq \mathtt{a}(\mathtt{n} - 1 - \mathtt{k})$. To create multiple transitions with a shared guard, activate the transition tool (↳) and click first on the border of `loop` to set the source situation, and then click outside of `loop` but inside the procedure box. This creates a transition ending in a *choice point* (rendered as a white square with a blue outline), which can serve as a starting point for further transitions. While both situations and choice points can serve as transition origins, choice points differ from situations in the two important respects. Firstly, a choice point must have exactly one incoming transition. Consequently, choice points can add new branches to a transition, but not join existing transitions. Secondly, choice points retain all assumptions from the start situation and guards and assignments on the transition path up to the choice point.

Add the guard $[\mathtt{k} < \mathtt{n} - 1 - \mathtt{k}]$ to the new transition. Since the guards from `loop` are now complete, we enable liveness checking for `loop` (right-click on `loop` and select Enable liveness check). The program should now look as follows (notice the solid squares at the transitions originating from `loop`, indicating that liveness is now checked for `loop`):


Java - ibptest/palindrome.ibp - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run View Window Help
Java
Packa
ibptest
palindrome.ibp
/home/joheriks/runtime-IBPE/ibptest/palindrome.ibp
Palette
Select
Marquee
Procedure
Declaration
Situation
Precondition
Postcondition
Transition
palindrome.ibp - ibptest

```
importing ds@parray
n:nat

palindrome [ a:parray[n,int]
             b:result bool   ]
k: pvar int

b = (forall (i:nat): i<n => a(i)=a(n-1-i))

[k>=n-1-k]
b:=true

loop
0<=k and k<=n-k
forall (i:nat): 0<=i and i<k => a(i)=a(n-1-i)

k:=0

[k<n-1-k]
```

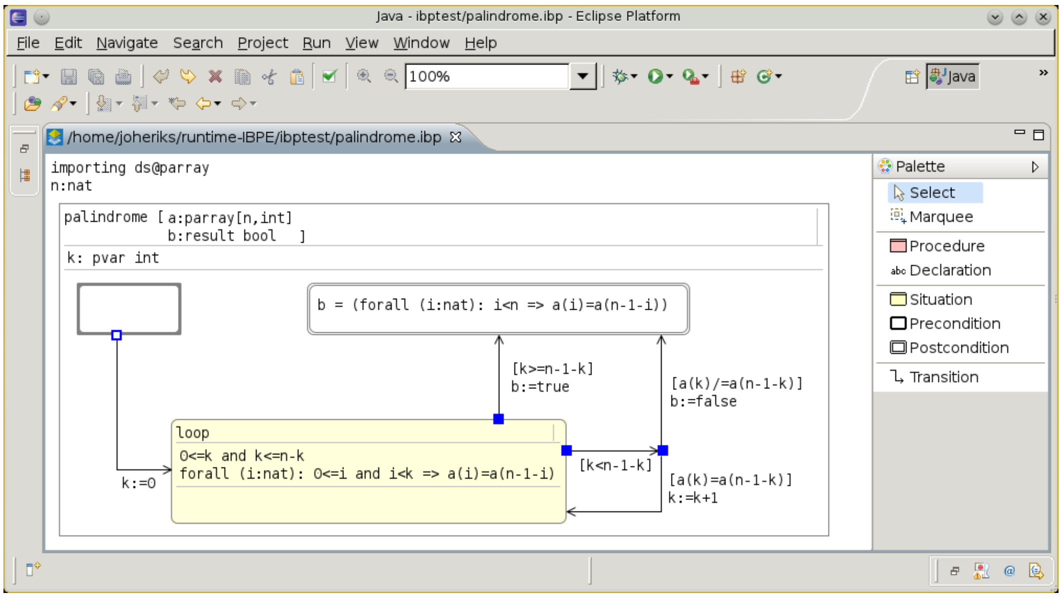
Check the program. Socos reports that all transitions are consistent and that the `loop` situation is live. As expected, a warning is issued about the choice point, since there are no outgoing branches from it yet. To resolve this issue, we first add the exit transition, starting from the choice point and ending at the postcondition. We then add the following statements to the exit transition:

```
[a(k)/=a(n-1-k)]
b:=false
```

Next, add the loop transition, starting from the choice point and ending at the `loop` situation. Add the following statements to the loop transition:

```
[a(k)=a(n-1-k)]
k:=k+1
```

We should also enable liveness checking at the choice point (right-click, Enable liveness check). The program should now look as follows:

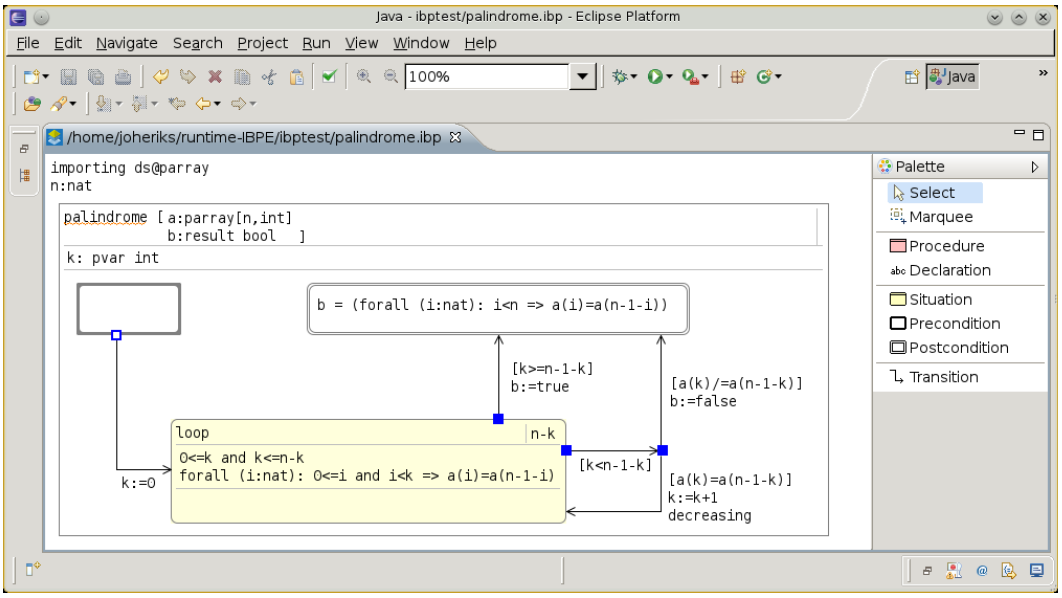

Java - ibptest/palindrome.ibp - Eclipse Platform
File Edit Navigate Search Project Run View Window Help
100%
Java
/home/joheriks/runtime-IBPE/ibptest/palindrome.ibp
Palette
Select
Marquee
Procedure
Declaration
Situation
Precondition
Postcondition
Transition

```
importing ds@parray
n:nat

palindrome [ a:parray[n,int]
             b:result bool   ]
k: pvar int

b = (forall (i:nat): i<n => a(i)=a(n-1-i))

[k>=n-1-k]
b:=true

[a(k)/=a(n-1-k)]
b:=false

loop
0<=k and k<=n-k
forall (i:nat): 0<=i and i<k => a(i)=a(n-1-i)

k:=0

[k<n-1-k]

[a(k)=a(n-1-k)]
k:=k+1
```

When we ask Socos to check the program, it now proves the added transitions consistent and the choice point live. It also issues a warning for termination. This is because we have added a loop but not yet given the termination function for `loop`. A termination function is a function from the program variables to the natural numbers that is decreased by the loop transition. The termination function must be declared in the upper right-hand corner of the `loop` situation: right-click on the corner and select Add declaration or, alternatively, activate the Declaration tool (abc) in the tool palette and then click on the corner. Type the following termination function: `n` − `k` (the lower bound 0 is implicit).

Finally, Socos also needs to know which transition we intend to decrease the termination function. Add the statement `decreasing` to the end of the loop transition. Socos now generates an additional verification condition that the annotated transition decreases the termination function. The final program is shown below:

Invariant diagrams are not restricted to single-entry constructs, but can be arbitrary graphs. The programmer must specify which transitions decrease the termination function. Socos issues an error if there are not enough transitions annotated with `decreasing` to cut all cycles in the diagram.


Java - ibptest/palindrome.ibp - Eclipse Platform
File Edit Navigate Search Project Run View Window Help
100%
Java
/home/joheriks/runtime-IBPE/ibptest/palindrome.ibp
Palette
Select
Marquee
Procedure
Declaration
Situation
Precondition
Postcondition
Transition

```
importing ds@parray
n:nat

palindrome [ a:parray[n,int]
             b:result bool   ]
k: pvar int

b = (forall (i:nat): i<n => a(i)=a(n-1-i))

[k>=n-1-k]
b:=true

[a(k)/=a(n-1-k)]
b:=false

loop                                        n-k
0<=k and k<=n-k
forall (i:nat): 0<=i and i<k => a(i)=a(n-1-i)

k:=0

[k<n-1-k]

[a(k)=a(n-1-k)]
k:=k+1
decreasing
```

When checking the program, Socos now reports the status COMPLETE. Hence we can conclude the `palindrome` procedure is consistent, live and terminating.

4. Updates and bug reports

Socos editor. Updates to the Socos editor are issued through the Socos update site:

```
http://www.imped.fi/update
```

To check for updates in Eclipse, first ensure that this site occurs in the list of update sites (Preferences ▷ Install/Update ▷ Available Software Sites). Then run Help ▷ Check for Updates. When a new version of Socos becomes available, Eclipse will notice it and ask to install it, replacing the previously installed version.

Socos checker. Updates to the Socos checker will be made announced on the main Socos page when available:

```
http://www.imped.fi/socos
```

New versions must be downloaded and installed manually. Older versions are currently available on request (contact `joheriks@abo.fi`).

Bug reports. Socos is a research prototype currently under development, and is likely to contain bugs. Discovered errors should be reported by email to `joheriks@abo.fi`. Please be as specific as possible in your bug report, include a description of how to reproduce the error. Your report should mention whether you are using local or remote checker, and if possible attach an `.ibp` file that demonstrates the bug. We also greatly appreciate any feedback on the tool to the above email address.

A. Language quick reference

This tutorial covers only a small subset of the PVS language. The full language reference is available at `pvs.csl.sri.com/doc/pvs-language-reference.pdf`.

Description	*Logic*	*PVS*
existential quantification	$(\exists x \in X \bullet P)$	`exists (x:`X`):`P
universal quantification	$(\forall x \in X \bullet P)$	`forall (x:`X`):`P
lambda abstraction	$(\lambda x \in X \bullet E)$	`lambda (x:`X`):`E
implication	$P \Rightarrow Q$	P `=>` Q P `implies` Q
equivalence	$P \Leftrightarrow Q$	P `<=>` Q P `iff` Q
conjunction	$P \wedge Q$	P `and` Q
disjunction	$P \vee Q$	P `or` Q
negation	$\neg P$	`not` P
equal to	$X = Y$	$X = Y$
not equal to	$X \neq Y$	X /= Y
less than or equal to	$X \leq Y$	X `<=` Y
greater than or equal to	$X \geq Y$	X `>=` Y
function application	$F(E_1, \ldots, E_n)$	F`(`E_1`,...,`E_n`)`
array access	$A[E]$	A`(`E`)`
array update	$A[I \leftarrow E]$	A `with [`I`:=`E`]`

Table 2: Expressions

Description	*Logic*	*PVS*
natural numbers	$\mathbb{N}$	`nat`
integers	$\mathbb{Z}$	`int`
positive integers	$\mathbb{N}_+$	`posnat`
rational numbers	$\mathbb{Q}$	`rational`
real numbers	$\mathbb{R}$	`real`
truth values	$\mathbb{B}$	`bool`
array (size $n \in \mathbb{N}$, type T)	$\mathsf{array}[0..\mathsf{n}-1]$ of T	`parray`$[n,T]$

Table 4: Types

Description	*Syntax*	*Example*
assignment	$Id_1, \ldots, Id_n := E_1, \ldots, E_n$	`x,y,z:=1,2,3`
assumption (guard)	$[E]$	`[x<y]`
procedure call	Id ($\langle E_1, \ldots, E_n \rangle$)	`p(a,b,x+y)`
decreasing annotation	`decreasing` $\langle Situation \rangle$	`decreasing S`

Table 6: Statements

Description	*Syntax*	*Context*	*Example*
PVS strategy	`strategy` ”$S - expr$”	Global	`strategy "(grind)"`
theory import	`importing` $Id@Id[Params]$	Global	`importing ds@parray[n,int]`
constant	$Id_1, \ldots, Id_n : X$	Global, signature, body	`n:int`
result parameter	$Id_1, \ldots, Id_n :$ `result` X	Signature	`y:result bool`
value-result parameter	$Id_1, \ldots, Id_n :$ `valres` X	Signature	`n,m:valres nat`
local variable	$Id_1, \ldots, Id_n :$ `pvar` X	Body	`a,b,c:pvar int`

Table 8: Declarations