Invariant Based Programming

Ralph-Johan Back
Abo Akademi and TUCS

June 2006

CREST
CENTRE FOR RELIABLE SOFTWARE TECHNOLOGY

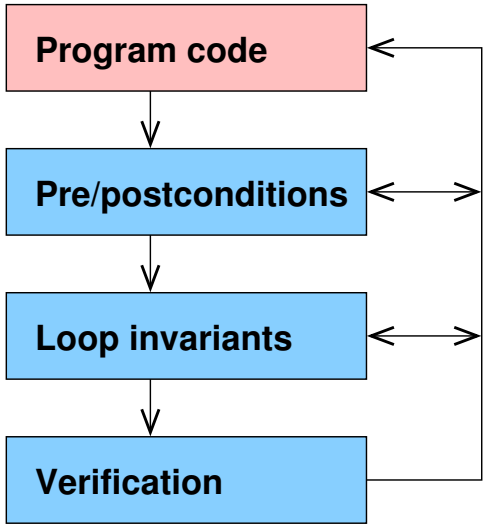
Constructing correct programs: alternative approaches

- **A posteriori correctness proof** (Floyd, Naur, Hoare, ...). Prove correctness after program has been written and debugged.

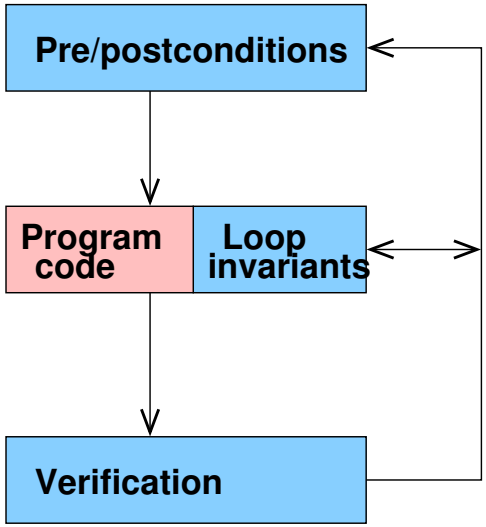
- **Constructive proofs** (Dijkstra, ...). Construct the program and its proof hand in hand, to satisfy given pre- and postconditions.

- **Invariant based programming** (Dijkstra?, Reynolds, van Emden, Back, ...). Formulate the program invariants first, then construct code that maintains these invariants. (Hehner has similar idea, but starts from relations rather than predicates)

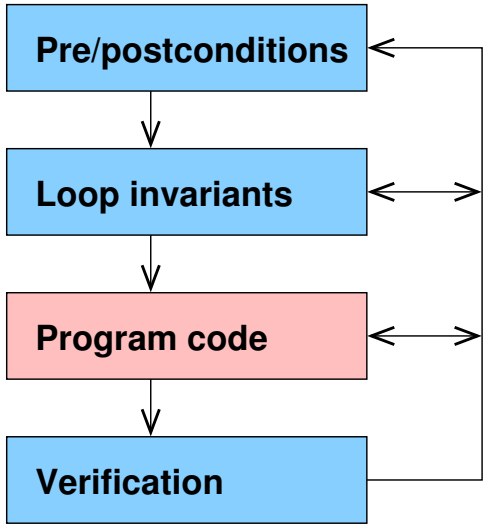
Place of coding in workflow



A posteriori proof



Constructive approach



Invariant based programming

Ivariant based programming: issues

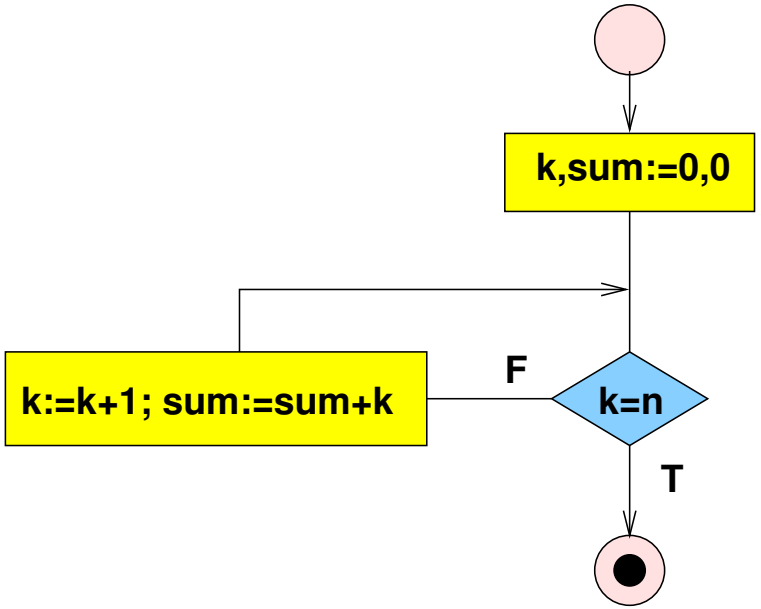
- What is a good notation for invariant based programs
- What is the work flow in constructing invariant based programs. In particular, how do you identify and formulate the invariants
- Comparing invariant based programs to ordinary programs
- Tools for invariant based programming
- Teaching invariant based programming

Notation: Nested invariant diagrams

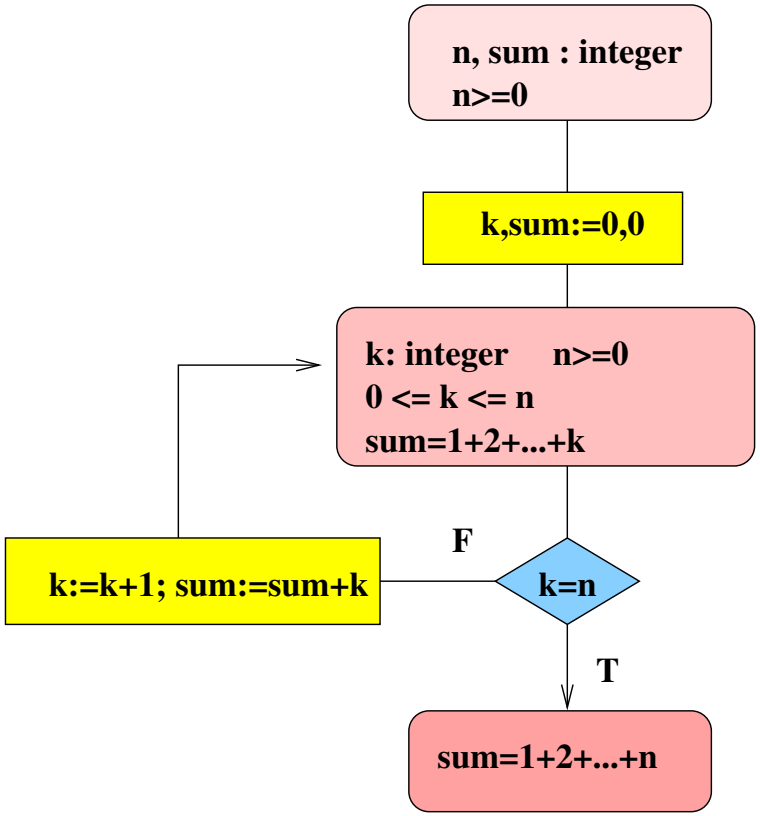
Structuring code vs structuring invariants

- Traditionally, program structure is based on the code
  - structured programming with single entry, single exit constructs
  - program modules, like procedures and classes
  - Invariants have to adapt to code structure
- For invariant based programming, we need to structure the invariants rather than the code
  - code then has to adapt to the invariant structure
  - single-enty single exit constructs not neccessarily the right structure

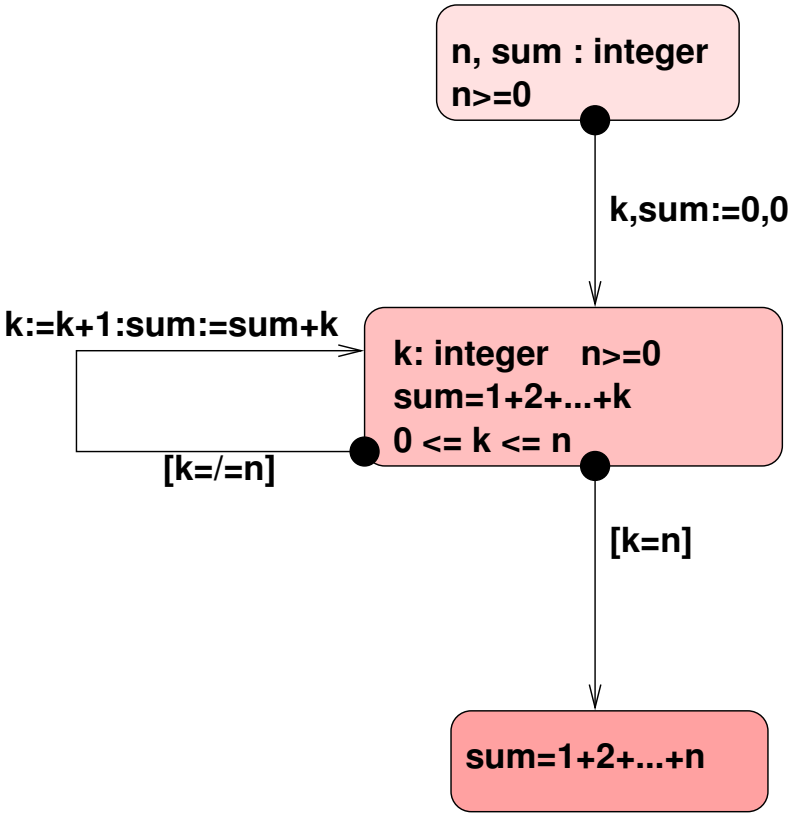
Summation program as a flowchart



Indicate situations explicitly



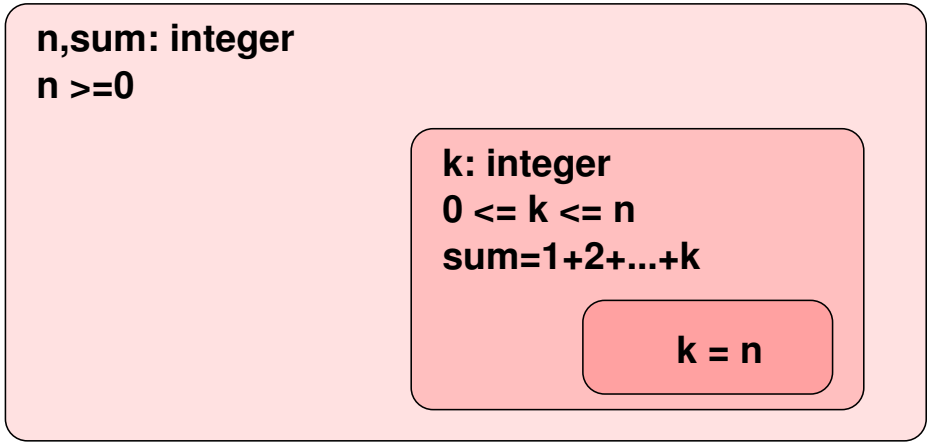
De-emphasize program statements



Structuring the invariants

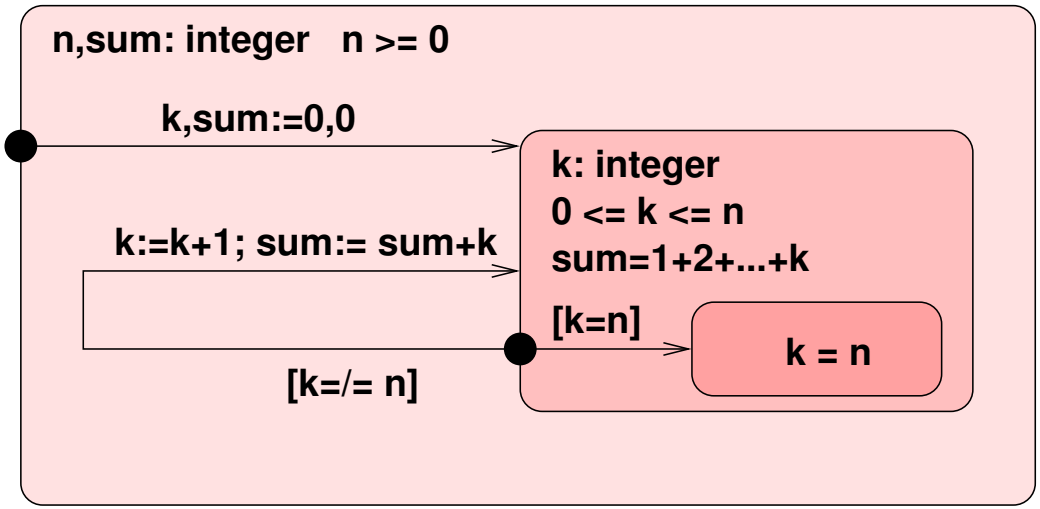
- A situation describes a set of states (the set of states that satisfies the condition for the situation).
- A situation is typically strengthened by adding new constraints.
- This gives a subset of the original situation (the subset where the additional constraints are also satisfied).
- We can use Venn diagrams to describe this strengthening of situations.

Nested invariants



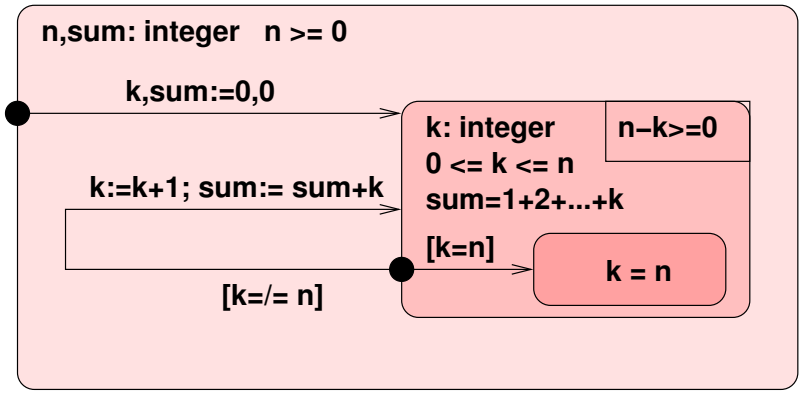
- We have omitted conditions that are implicit because of nesting. For instance, $n \geq 0$ is written only once in the outermost situation, but it will hold in all nested situations.

Nested invariant diagram



- The statement arrows between the situations are the same as before.
- This diagram is equivalent to the previous ones, except that we have nested the situations

Indicating termination



- We write the termination function inside the box in the right upper corner of the invariant.

- It shows here that $n - k \geq 0$ must hold in the indicated situation, and that the function $n - k$ must be decreased before re-entering this situation.

Nested invariant diagrams and state charts

- Nested invariant diagrams are similar to *state charts*. Both are extensions of state transition diagrams. However, interpretation and intended use is different.
- State charts specify the control flow in reactive systems, without any concern for correctness, whereas invariant diagrams are essentially correctness proofs.
- A state chart is usually an abstraction of a larger software system, whereas invariant diagrams describe the whole program.
- Invariant diagrams can be extended with features from state charts that are deemed useful (like product states, and signals/procedure calls).

Workflow: by example

Case study: sorting

Problem: Sort an array of integers into non-decreasing order.

The algorithmic solution

- We consider the simplest possible sorting program, insertion sort.
- Essentially, we sort the array by moving a cursor from left to right in the array.
- At each stage we find the smallest element to the right of the cursor, and exchange this element with the cursor element.
- After this, we advance the cursor, until we have traversed the whole array.

Identifying the different situations

- We draw figures that illustrates the basic data structures involved and how they will be changed during execution of the algorithm.
- We can also hand simulate the execution of the algorithm with concrete data.
- From the figures and hand simulation, we then try to identify
  - the intial situation,
  - the final situation(s), and
  - recurring situations and intermediate goals,

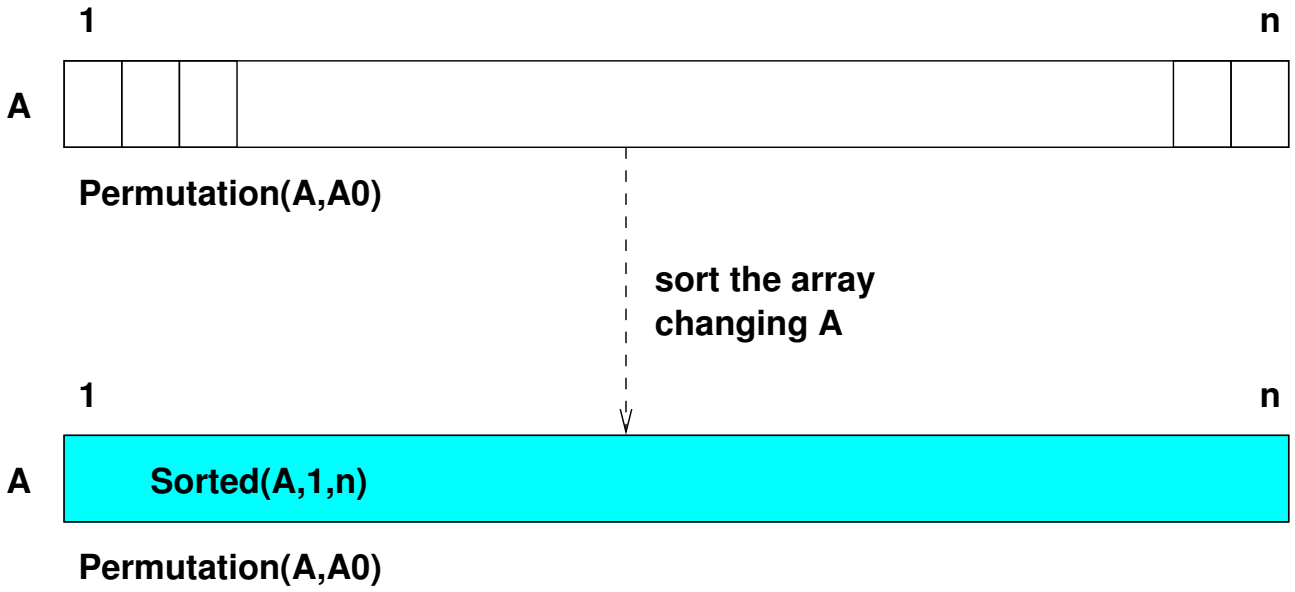
  and express these generally.
- We may need to adjust these situations later, when constructing and checking the transtitions.

A little domain theory

- $Sorted(A, i, j)$ means that the array elements are non-decreasing in the (closed) interval $[i, j]$,
- $Partitioned(A, i)$ means that every element in array A below index i is smaller or equal to any element in A at index i or higher, and
- $Permutation(A, A0)$ means that the elements in array A form a permutation of the elements in array $A0$.

Initial and final situation

We identify the initial situation and the required final situation



Precise description

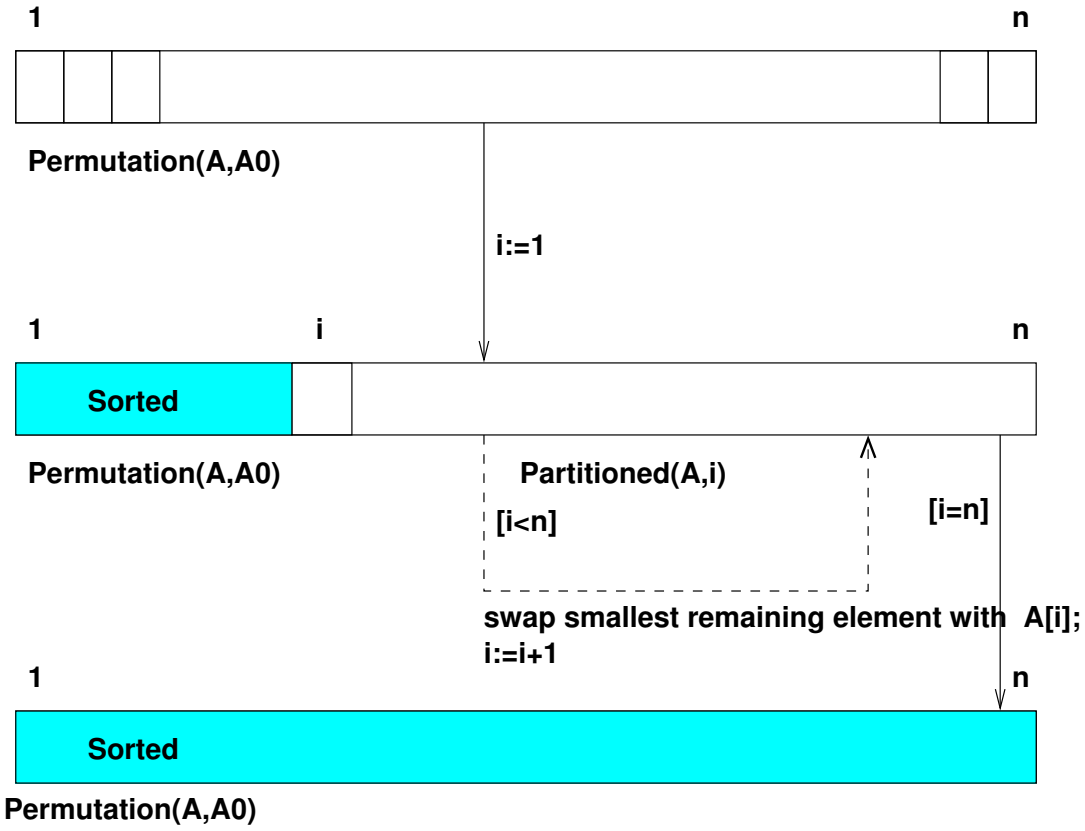
Initially

$$
\begin{aligned}
& n : integer\, n \geq 1 \\
\wedge \quad & A : array\, 1 : n\, of\, integer \\
\wedge \quad & Permutation(A, A_0)
\end{aligned}
$$

Finally

$$
\begin{aligned}
& Sorted(A, 1, n) \\
\wedge \quad & Permutation(A, A_0)
\end{aligned}
$$

Intermediate situation



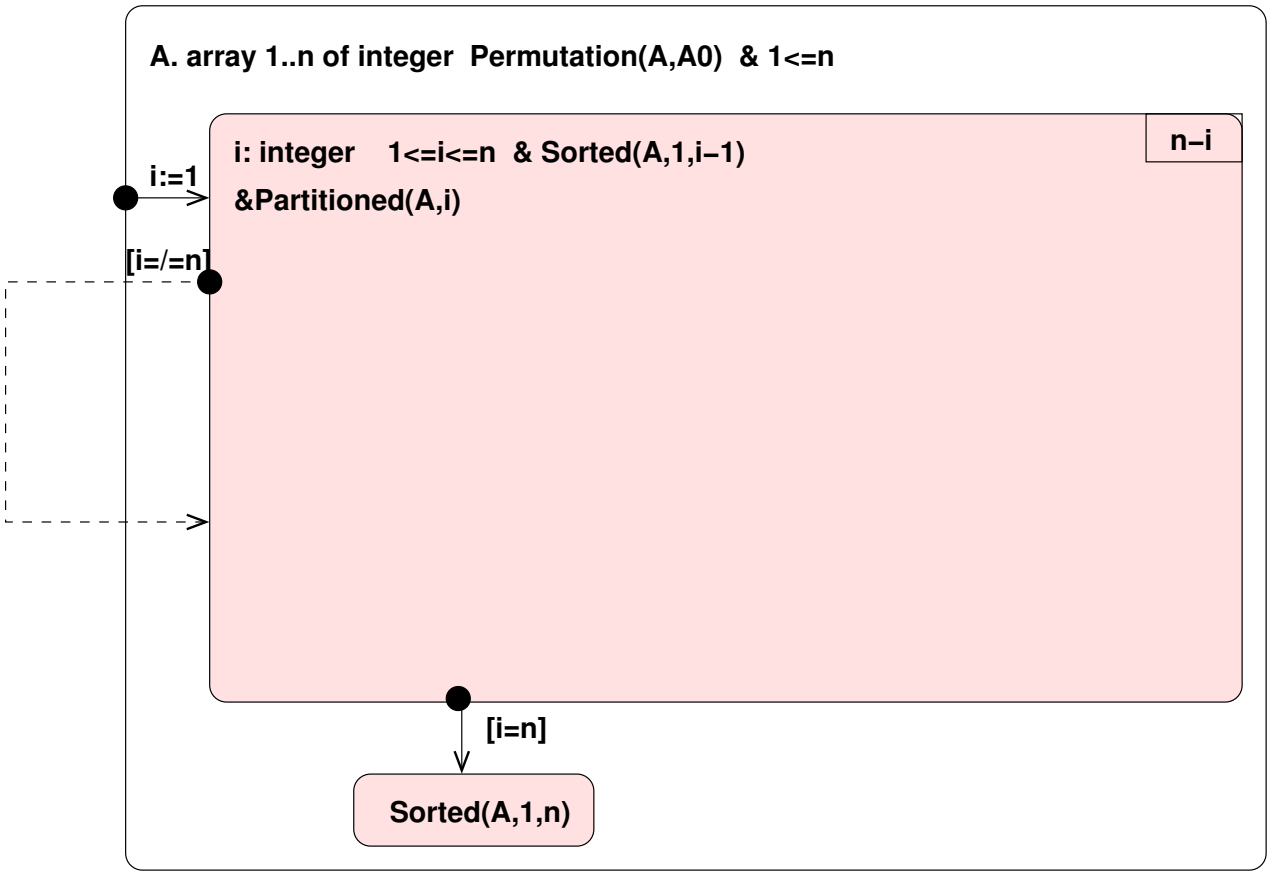
Precise description

The intermediate situation adds the following conditions to the initial situation

$$i : integer \wedge 1 \leq i \leq n \wedge Sorted(A, 1, i-1) \wedge Partitioned(A, i)$$

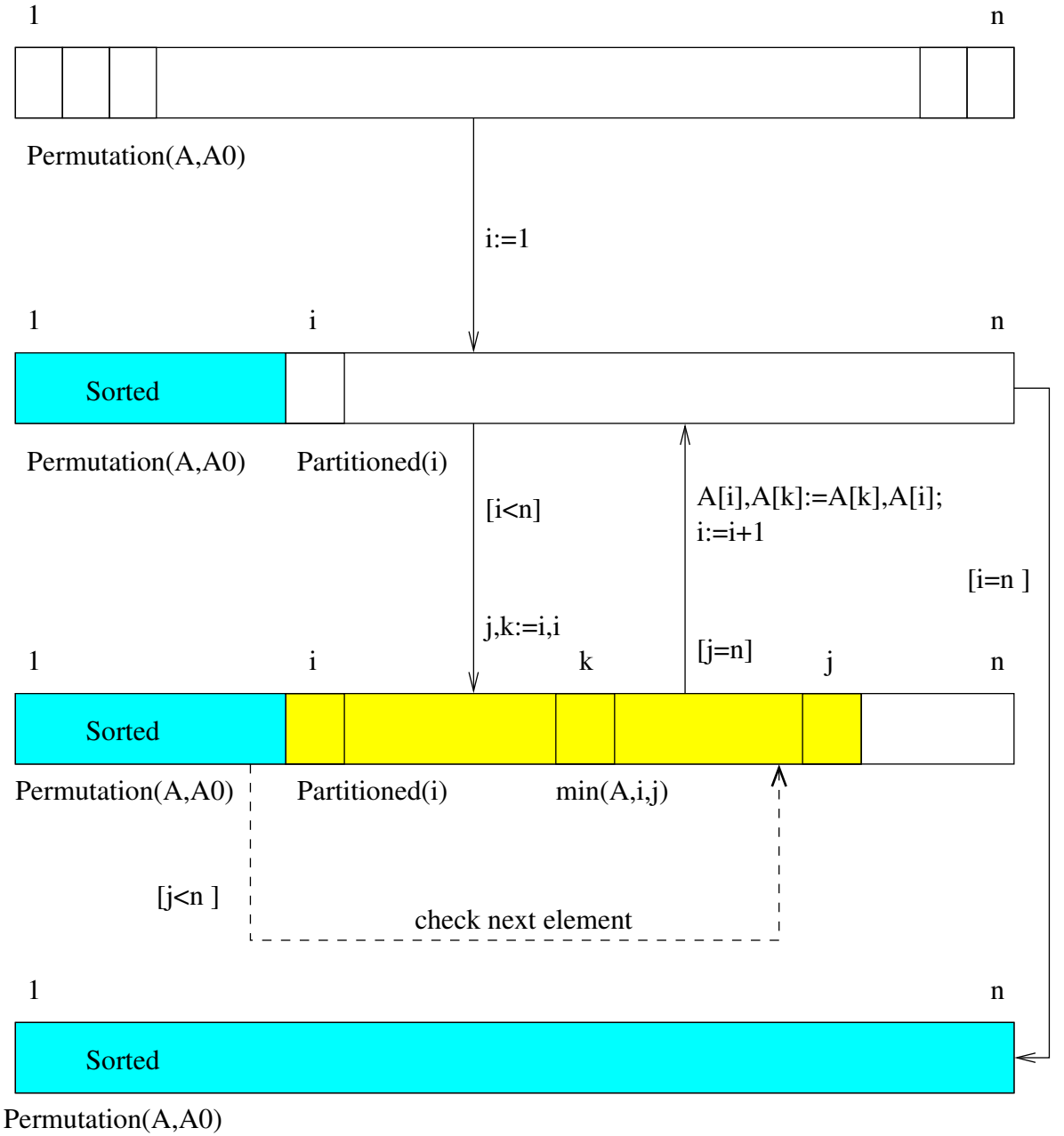
- The assignment $i := 1$ will establish this intermediate situation.
- Condition $i = n$ will imply the final situation.
- Remains to figure out how to make progress towards this condition while maintaining the intermediate situation.

Invariant diagram



Identifying smallest element

- To find the smallest remaining element, we need to scan over all the remaining elements
- We need a loop here also.
- We add a fourth situation, where part of the unsorted elements have already been scanned for the least element.


1
n
Permutation(A,A0)
i:=1
1
i
n
Sorted
Permutation(A,A0)
Partitioned(i)
[i<n]
A[i],A[k]:=A[k],A[i];
i:=i+1
[i=n]
j,k:=i,i
[j=n]
1
i
k
j
n
Sorted
Permutation(A,A0)
Partitioned(i)
min(A,i,j)
[j<n]
check next element
1
n
Sorted
Permutation(A,A0)

Precise description

- The new situation is characterized by the additional constraints

$$k, j : integer \wedge i \leq k \leq j \leq n \wedge A[k] = min\{A[h] | i \leq h \leq j\}$$

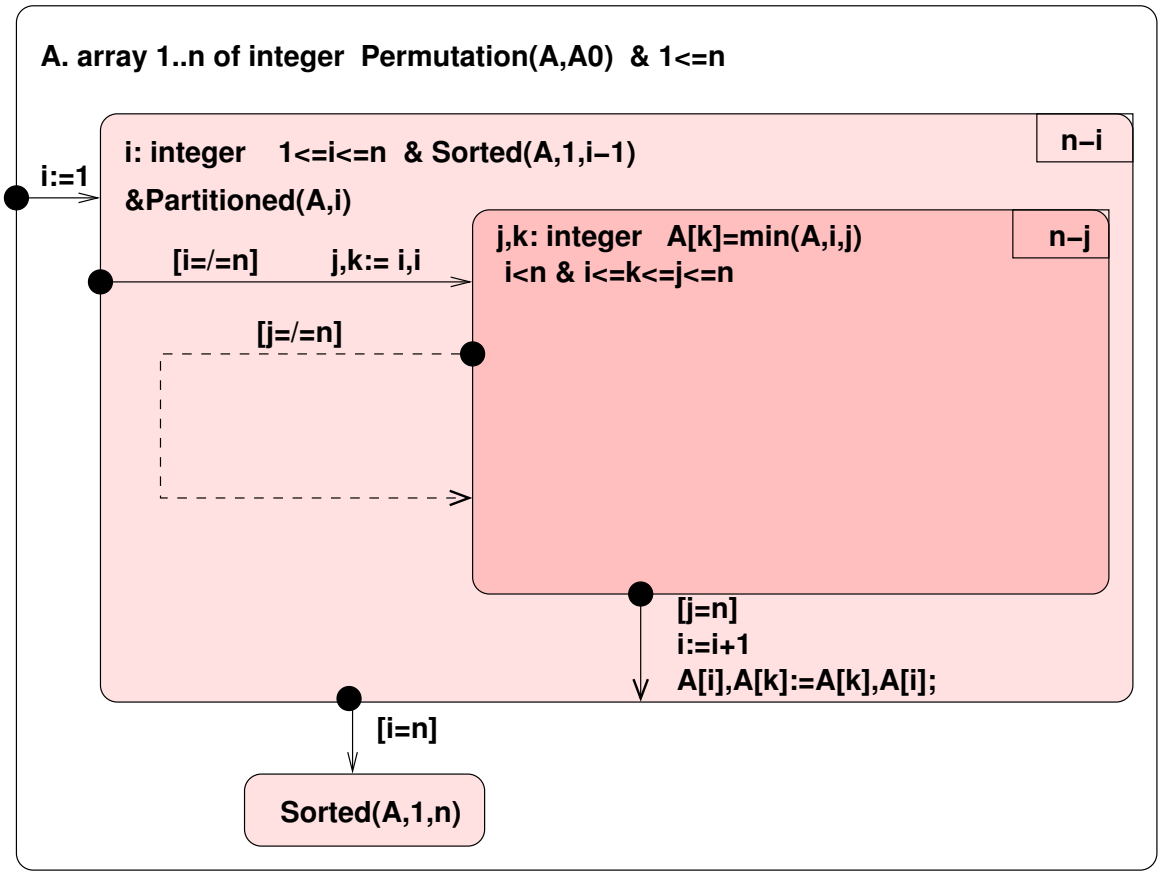
- We check that this situation is established from the previous intermediate situation by the assignment $j, k := i, i$ when $i \neq n$.

- We also check that if $j = n$, then

$$A[i], A[k] := A[k], A[i]; \; i := i + 1$$

will establish the first intermediate situation, as indicated in the diagram.

Invariant diagram



Preserving the invariant

- Finally, we need to check that the second invariant is preserved while making progress. We need to show that when $j \neq n$, the statement

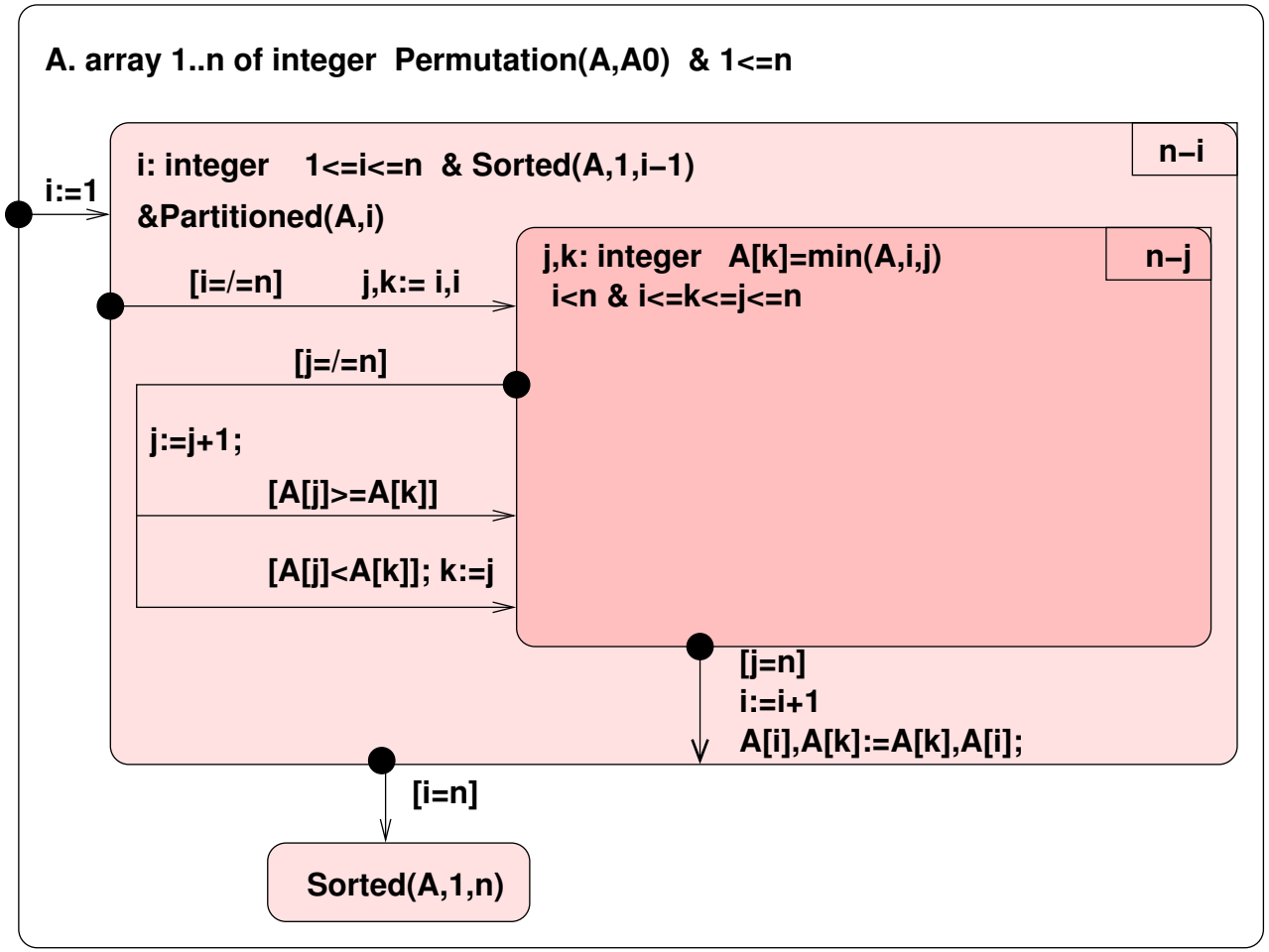
$$j := j + 1; \text{if } A[j] < A[k] \text{ then } k := j \text{ fi}$$

  preserves the second invariant.

- The inner loop will eventually terminate because $n - j$ is decreased but is bounded from below.

- The outer loop will terminate because $n - i$ is decreased and is bounded from below.

Complete invariant diagram

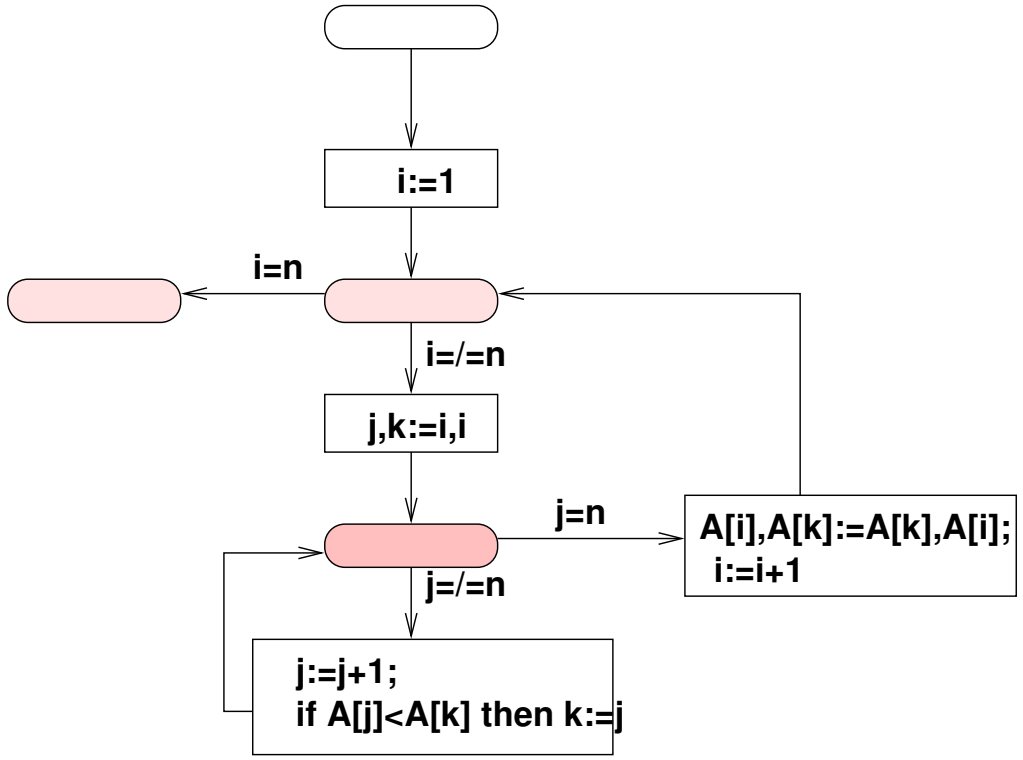

A. array 1..n of integer Permutation(A,A0) & 1<=n
i:=1
i: integer 1<=i<=n & Sorted(A,1,i−1)
&Partitioned(A,i)
n−i
[i=/=n]
j,k:= i,i
j,k: integer A[k]=min(A,i,j)
i<n & i<=k<=j<=n
n−j
[j=/=n]
j:=j+1;
[A[j]>=A[k]]
[A[j]<A[k]]; k:=j
[j=n]
i:=i+1
A[i],A[k]:=A[k],A[i];
[i=n]
Sorted(A,1,n)

- Outermost situation gives the background assumptions for the algorithm.
- The nested situation is what holds when we have sorted the array up to $i - 1$, but have not yet started to scan for the smallest element in the rest of the array.
- Innermost situation holds while we are scanning for the least element.
- The second nested situation is the final situation.

Comment

- We could also have nested the final situation inside the first invariant.
- However, that would have indicated that we also had some information about the value of i at exit.
- As this is not needed, we prefer to keep this invariant nested only inside the initial situation.

Flow chart description



Using figures

- We used figures quite extensively in the derivation of the invariant diagram. They are needed to understand the algorithm intuitively.
- However, a figure describes only a special case.
- The figures allow us to read out the logical formulation of the invariants in a rather straightforward way.
- The invariant diagrams describe the general case. They should therefore be used when checking corrctness.

Invariant based programs vs ordinary programs

Correctness issues

An invariant based program is

- *consistent*, if all invariants (predicates) are preserved
- *terminating*, if there are no infinite execution loops, from any invariant
- *live*, if termination only occurs at final situations

An invariant based program is *totally correct*, if it is consistent, terminating and live

Establish correctness as a side effect

- Standard programming practices produce program code, that needs to be tested and debugged. Does not guarantee sufficiently high reliability. Verification could be done a posteriori, but is in practice not attempted, because
  - the problem has already been solved (kind of), there is already something that works and can be delivered
  - too much additional effort to find invariants and prove verification conditions
  - lack of powerful tools for checking correctness automatically
- With invariant based programming, we get the invariants for free, as part of the programming process. Does require more effort, but promises to give a much higher reliability

Role of figures

- Any programmer who is solving a problem like the one above will (or at least should) draw the kind of figures shown above, to get a feeling for how the program should behave.
- In invariant based programming, these figures are preserved as the invariants. In ordinary programming, they are usually lost in subsequent steps.
- By elevating these figures to a more distinguished position in program construction, we are more likely to preserve them and keep them up to date.
- Role of figures in program construction is similar to role of figures in physics and engineering.

Correctness notion

The correctness notion for invariant based programs is stronger than the traditional notion.

- Traditional correctness: a program started in an initial state that satisfies the pre-condition must terminate in a final state that satisfies the postcondition.

- Any situation can be an initial state in an invariant based program. Correctness must then also holds for states in interior situations that cannot even be reached by a legal execution from an initial situation.

- Correctness of an invariant based program is also stronger in that all interior situations must be satisfied during execution. It is not sufficient that only the final situations are satisfied upon termination.

Is the correctness notion too strong

- One could argue that the correctness requirement for invariant based programs is too strong, that the traditional program correctness notion should be enough.

- However, as soon as a program has at least one loop, an inductive argument is needed for the correctness proof. If this argument takes the form of a proof with invariant assertions, one will in fact end up establishing the correctness of the program as an invariant based program.

- In other words, usual proof techniques also establish a stronger correctness property for a program than what is required by the traditional notion of correctness.

Locality

- One can *reason about* invariant based programs in a local fashion. Can consider each situation at a time, together with the situations that can be reached from this situation with simple transitions. Other situations can be ignored.

- One can *construct* an invariant based program in a local fashion, one situation or transition at a time. Similar to functional programming and logic programming. Provides a lower level of modularity for imperativ programs than what is provided by procedures.

- One can *change and fix* an invariant based program in a local fashion.

- Only termination requires an global view of the program: one needs to check that

each possible cycle in the diagram decreases some termination function, which is not increased anywhere in the program.

Programming methodology

- The execution model of an invariant based program is quite general, we allow for failure, (miraculous) success, infinite execution and normal termination.
- The minimum requirement for an invariant based program is that it is consistent. The program does not have to terminate or be live.
- Liveness may not hold because we have not (yet) covered all possible cases for some internal situation, i.e., there are cases for which we still need to provide transitions. However, the program constructed thus far is consistent (although incomplete).
- Similarly, we may have a consistent program, but we have yet to tackle the termination of the program, which may require redefinition of some invariants.

Stepwise development

An invariant based program is constructed by a sequence of successive development steps, where each step preserves the consistency of the previous step (liveness and termination may vary during development)

- Add some new situations (initial situation, intermediate situations or final situations) or some new transitions to a situation, to increase liveness of the program.
- Modify or remove some transitions or situations
- A development step may also change the program in order to improve termination or liveness

Checking consistency

- Invariant based programming requires that we carefully check the consistency of each transition when it is introduced.
- Leaving the consistency checks to a later stage in program development will only accumulate errors in the program and make the consistency checking too laborious. We must remove the errors when they are introduced, not sometimes later
- It will also decrease the motivation for carrying out consistency checks at all, because too many interdependent things then need to be considered and changed.
- Consistency checks can be done at different levels of rigour.

Tools for invariant based programming

The Socos Environment

- The Socos environment supports invariant based programming
- Provides a graphical and textual representation of invariant based programs
- Uses theorem provers to automatically discharge verification conditions. For the moment, we use Simplify and PVS for this. Socos only shows proof obligations that have not been proved automatically.
- Environment compiles invariant based program directly to Python; executes them, has a debugging mode
- Can check procedure pre- and postconditions and invariants during run time

Work in progress

- Working on a simple semantics for invariant based programs
- Extending the approach to procedures, modules and classes, with recursion, inheritance/superposition and implementation/data refinement
- Improving the automatic verification and user friendliness of Socos
- Experimenting with constructing larger, modularized invariant based program

Teaching invariant based programming

Experiments with invariant based programs

- I have been trying this approach in a number of small sessions, usually with two persons constructing an invariant based program together (some 10 experiments, 2-3 hours per session)
  - Works without problems, both for IFIP WG2.3 members and novices to program verification
  - Finding initial invariants is quite easy, when one starts from a figure
  - Invariant is improved when transitions are introduced one by one
  - Some very subtle bugs are found in transitions/invariants during verification
  - Tool support for automatic checking highly desirable (but we can live without it for small programs)

Work flow: left to right / top to bottom

	figures	extend theory	formulate	extend diagram	check
initial situation	X	X	X	X	
final situation	XX	XXX	XX	X	
intermediate sit	X	XXX	XX	X	
entry transition			X	X	X
exit transition			X	X	X
iteration trans			XX	X	XX
termination		X	X	X	X
liveness			X		X
difficulty	medium	can be hard	medium/easy	easy	medium

Teaching invariant based programming

- Have taught a course on invariant based programming for our Ph.D. students
- Approach has also been tried in a course at technical college level (combining structured derivations and invariant based programming)
- Planning a programming course for first year CS students using invariant based programming

Teaching experiment at Åbo Akademi

We will teach two basic courses to first year CS students:

- Using logic in mathematical reasoning (Back and von Wright: Mathematics with a little bit of logic). Supports formulating invariants in practice and proving verification conditions

- Invariant based programming. Based on previous course, applies logical reasoning to building correct programs.

This will give programming a standard mathematical basis, which also can be used in practice to build programs. In addition, we also teach traditional programming courses, with a chosen programming language, and using the code-test-debug cycle.

Where to teach programming

We can teach programming at different levels in the education system:

- High school. The above mentioned courses can be taugh in high school. Should fit into a mathematics curriculum (which is expanded to include CS)

- Polytechnics. First year courses, provides a foundation for programming.

- Universities. First year courses, teaching the basic foundation of programming. Continue with techniques for constructing larger programs that are correct by construction.

Advantage

- The above two courses would be quite similar in level and difficulty to other math and physics courses in high school.
- We teach students a method for programming that allows them to check themselves that the program works correctly.
- The course in logic improves our ability to reason about programs and mathematical derivations in general.
- We could teach invariant based programming as a first programming course
- Satisfaction of building something solid, rather than uneasy feeling of undetected errors.

Thank you

More details in report at my web pages: www.abo.fi/~backrj