Verifikation nebenläufiger Programme
Wintersemester 2004/05

Ulrich Hannemann Jan Bredereke

20 A Proof System for Sequential Programs

Using the terminology of the verify-while-develop paradigm, the proof methods developed until now for the systematic derivation of systems from their specifications provide no help once specifications of basic transition diagrams are obtained. This is unacceptable, for such basic diagrams correspond to sequential programs or processes, and the systematic derivation of such programs from their specifications constitutes one of computer science's major tasks. Solutions to this problem have been developed under the heading of "structured programming", especially by Edsger W. Dijkstra and his group of colleagues since the end of the sixties. Consequently, in this Session we extend the methods developed previously to sequential program operators.

What was missing also in the previous chapters was *a direct link between program verification and the textual representation of programs.*

A solution to these problems is provided by (sequential) *Hoare logic* [Hoa69]. First, this logic is compositional w.r.t. its sequential programming operators and, therefore, extends the compositional methods developed for concurrency in the previous chapters to sequential programming. Second, Hoare logic provides an immediate link between program verification and program text.

20.1 Syntax and Informal Meaning of GCL$^+$ Programs

The programming languages in these chapters are based on classes of expressions e, and boolean expressions b. These two types of expression are used also within the assertions of Hoare formulae. Their syntax is given in Table 1.

Table 1: Syntax of expressions and boolean expressions.

Expression	$e ::=$	c	x	$(e_1 + e_2)$	$(e_1 - e_2)$	$(e_1 * e_2)$
Boolean Expression	$b ::=$	$e_1 = e_2$	$e_1 < e_2$	$(\neg b)$	$(b_1 \wedge b_2)$	

The programming language is based on Dijkstra's guarded-command language GCL [Dij76]. Due to its elegant mathematical flavour, this language is well suited for the construction and verification of sequential algorithms. A typical aspect of GCL is that nondeterminism is an essential aspect of the language constructs for conditionals and for loops. These nondeterministic counterparts of the **if** $-$ **then** $-$ **else** and **while** constructs are especially useful when developing algorithms.

Let *Pvar* be a nonempty set of program variables, and *VAL* be a domain of values. Let $x \in \mathit{Pvar}$, and $c \in \mathit{VAL}$.

We have chosen fairly small classes of expressions; other arithmetical and boolean operations that can be seen as abbreviations will be used freely in examples. For instance, we use $b_1 \vee b_2 \overset{\text{def}}{=} \neg((\neg b_1) \wedge (\neg b_2))$. In examples we also often omit parentheses, relying on the usual precedence rules for arithmetical and boolean operators.

The syntax of GCL is given in Table 2.

Table 2: Syntax of the Programming Language GCL.

Statement	$S ::=$	$\bar{x} := \bar{e}$	$S_1 \,;\, S_2$	**if** $[]_{i=1}^{n} b_i \to S_i$ **fi**	**do** $[]_{i=1}^{n} b_i \to S_i$ **od**

Guarded commands were invented to express *sequential* programs, and atomicity of operations was therefore not an important issue. By the principle of separation of concerns, assignments and boolean tests are strictly separated, and not combined in one action of the form of a guarded assignment $\langle b \to \bar{x} := \bar{e} \rangle$, as is the case for transition systems. In the following chapters, where we study concurrency, we take in essence GCL as the point of departure, and add language constructs like parallel composition and communication commands. Here, atomicity is of foremost importance, and we need a language construct like the guarded assignment. Such guarded assignments form the basis of our transition systems, and also of many other formal methods for concurrency, such as the development styles due to Ralph Back, see, e.g., [Bac80] and [Bac90], and to Jay Misra & Mani Chandy [CM88]. Therefore, we introduce an extension, called GCL$^+$, that incorporates guarded assignments and, for instance, allows such guarded assignments at places where in GCL only boolean guards are allowed. GCL$^+$ as such will be used mainly in theoretical investigations concerning soundness and completeness results, and is therefore chosen such that it has a simple syntax and, more importantly, provides for a straightforward translation of GCL$^+$ programs to equivalent transition systems. The language constructs of GCL$^+$ correspond to natural composition operations for transition systems. All GCL$^+$ operations amount, on the level of transition systems, to taking the disjoint union of the component graphs, followed by identification of the appropriate entry and/or exit labels. For instance, the choice construct **if** $[]_{i=1}^{n} S_i$ **fi** at the level of transition systems amounts to taking the disjoint union of the graphs denoted by the components S_i, whereafter the entries for all components are merged into one entry for the whole, and similarly, all exit locations are merged into one exit (see Figure 3 on page 5). Observe that, on the level of transition systems, this expresses the nondeterministic union of $S_1, \ldots, S_n$, which is, indeed, the informal meaning of **if** $[]_{i=1}^{n} S_i$ **fi** . The loop construct in GCL$^+$ is somewhat less elegant than that of GCL, due to the fact that in transition systems the transition that terminates a loop must be given explicitly. Therefore, the loop construct in GCL$^+$ has the form **do** $S_B \; [] \; S_E$; **exit od**, including an explicit exit branch. Informally, when such a loop is executed, the guards of the initial transitions of the "body" S_B and the "exit branch" S_E are evaluated, and one of the transitions with a guard that evaluates to "*true*" is nondeterministically chosen. If some S_B transition is chosen, then the rest of S_B is executed and, upon reaching the exit location of S_B, the whole process is repeated; if some S_E transition is chosen, then after executing the rest of S_E and upon reaching the exit location of S_E the iteration terminates. Below we show how the loop construct from GCL$^+$ can be used to simulate the GCL loop and the more conventional **while – do** construct. One can also simulate a construct S^* which executes S zero or more times, where the actual number of iterations is chosen nondeterministically. (This is analogous to the Kleene star

Table 3: Syntax of the Programming Language GCL$^+$.

Statement	$S ::=$	$\langle b \rightarrow \bar{x} := \bar{e}\rangle \ \mid\ S_1 \,;\, S_2 \ \mid\ \textbf{if}\ \ []_{i=1}^{n} S_i\ \ \textbf{fi}\ \mid$ $\textbf{do}\ S_B\ []\ S_E\ ;\ \textbf{exit}\ \ \textbf{od}$

operator for regular languages). The intended meaning for S^* is captured by the GCL$^+$ statement **do** $S[]$**skip** ; **exit od**, where **skip** denotes the guarded assignment with guard "*true*" and an assignment to an empty list of variables.

The following restrictions are imposed. Let $\equiv$ denote syntactic equality.

- For an assignment $\bar{x} := \bar{e}$, with $\bar{x} \equiv (x_1, \ldots, x_k)$ and $\bar{e} \equiv (e_1, \ldots, e_k)$, for $i = 1, \ldots, k$, the variables $x_i \in Pvar$ must all be distinct. This also applies to the assignment part of a *guarded assignment* $\langle b \rightarrow \bar{x} := \bar{e}\rangle$.
- The statements S_i, S_B and S_E in **if** $[]_{i=1}^{n} S_i$ **fi** and **do** S_B $[]$ S_E; **exit od** must be *guarded*, where we say that S is guarded if
  - S is an action, that is, of the form $\langle b \rightarrow \bar{x} := \bar{e}\rangle$, or
  - S is of the form S_1 ; S_2, and S_1 is guarded, or
  - S is of the form **if** $[]_{i=1}^{n} S_i$ **fi** , and S_i is guarded, for $i = 1, \ldots, n$.

The meaning of the GCL$^+$ constructs is obtained by translating GCL$^+$ statements into transition systems, which is done in Section 20.2. The guardedness of S_i in **if** $[]_{i=1}^{n} S_i$ **fi** and of S_B, S_E in **do** S_B $[]$ S_E ; **exit od** is needed in order to define the semantics of these constructs in terms of transition systems as described below. We will come back to the issue of guardedness in the next section.

GCL statements are formally introduced as abbreviations of GCL$^+$ statements; their translation is straightforward:

- An assignment $\bar{x} := \bar{e}$ abbreviates $\langle true \rightarrow \bar{x} := \bar{e}\rangle$.
- We also need **skip** , which abbreviates a multiple assignment with an *empty* list of variables.
- The *guarded command* notation **if** $[]_{i=1}^{n} b_i \rightarrow S_i$ **fi** abbreviates

$$\textbf{if}\ \ []_{i=1}^{n} \langle b_i \rightarrow \textbf{skip}\ \rangle\ ;\ S_i\ \ \textbf{fi}\ .$$

- The guarded loop, **do** $[]_{i=1}^{n} b_i \rightarrow S_i$ **od** abbreviates **do if** $[]_{i=1}^{n} b_i \rightarrow S_i$ **fi** $[]$ $\langle \neg b_G \rightarrow$ **skip** $\rangle$; **exit od**, where $b_G \equiv b_1 \vee \cdots \vee b_n$.

Note that, according to our translation, a GCL guard is executed as a transition of its own, i.e., not combined with any assignment to program variables. For sequential programs this difference hardly matters; for concurrent programs this difference is crucial because of the possibility of interference between the guard and the assignment.

Observe that not only GCL programs, but also many other conventional programming constructs can be defined as an abbreviation:

- Sometimes, a boolean guard of the form b is used as a statement itself, which is then called a *filter*, and abbreviates $\langle b \to \mathbf{skip}\ \rangle$.
- As an alternative to the guarded command notation one also uses constructs in the style of ALGOL, PASCAL, ADA, etc. E.g., **if** b **then** S_1 **else** S_2 **fi** abbreviates **if** $b \to S_1 [\!] \neg b \to S_2$ **fi** and **while** b **do** S **od** abbreviates **do** $[\!]_{i=1}^{1} b \to S$ **od**, which is called a while statement.

In examples we will sometimes use boolean variables, ranging over $\{tt, ff\}$. Conventional abbreviations are used, such as $tt \stackrel{\text{def}}{=} 0 = 0$, $ff \stackrel{\text{def}}{=} \neg tt$, $b_1 \wedge b_2 \stackrel{\text{def}}{=} \neg(\neg b_1 \vee \neg b_2)$, etc.

20.2 Semantics of GCL$^+$- A Compositional Translation of GCL$^+$ to Transition Systems

The semantics of GCL$^+$ statements is defined by associating a transition system $T[\![S]\!]$ with program S. Recall that the difference between a transition system and a transition diagram is that a transition system is an equivalence class of transition diagrams, obtained by abstracting from the particular names of nodes of such diagrams, cf. Session 1. We use transition systems, but almost always define operations on such systems in terms of their representative transition diagrams. The main advantage of transition systems over transition diagrams is that operations such as sequential composition become associative by definition, since the semantics of a transition diagram does not depend on the particular names chosen for its nodes.

The translation from GCL$^+$ statements to transition systems is in a compositional style. The first step is to define composition operations for transition systems. All definitions below are given relative to a fixed state space Σ.

Definition 20.1 (Operations on transition systems)

- For a guarded assignment, we define $\mathbf{step}(\langle b \to \bar{x} := \bar{e}\rangle)$ as the transition system represented by $(\{s,t\}, \{(s, b \to \bar{x} := \bar{e}, t)\}, s, t)$.
- Let P_1 and P_2 be transition systems represented by (L_1, T_1, s, r) and (L_2, T_2, r, t), and assume that $L_1 \cap L_2 = \{r\}$. We define $P_1 ; P_2$ as the transition system represented by $(L_1 \cup L_2, T_1 \cup T_2, s, t)$.
- Let P_i be transition systems represented by (L_i, T_i, s, t), for $i = 1, \ldots, n$, where we assume that $L_i \cap L_j = \{s, t\}$, for $1 \leq i < j \leq n$.
  We define **if** $[\!]_{i=1}^{n} P_i$ **fi** as the transition system represented by
$$(\bigcup_{i=1}^{n} L_i, \bigcup_{i=1}^{n} T_i, s, t).$$
- Let P_B be a transition system represented by (L_B, T_B, s, r) and let P_E be a transition system represented by (L_E, T_E, s, t), such that $L_E \cap L_B = \{s\}$. We define **do** $P_B\ [\!]\ P_E$; **exit** **od** as the transition system represented by $((L_B \setminus \{r\}) \cup L_E, T'_B \cup T_E, s, t)$, where T'_B is like T_B except that transitions of the form $(l, b \to \bar{x} := \bar{e}, r)$ have been replaced by $(l, b \to \bar{x} := \bar{e}, s)$. $\square$

From the above definitions one can see why transition systems rather than diagrams are useful, because one can choose in every one of the above clauses representatives such that the constraints on the label sets L_i are satisfied. Now that we have operations on transition systems that match the language operations of GCL$^+$, it is easy to define the T function, which translates GCL$^+$ programs to transition systems:

Definition 20.2 (Semantics of GCL$^+$ statements)

- $T\,[\![\, \langle b \rightarrow \bar{x} := \bar{e} \rangle \,]\!] \stackrel{\text{def}}{=} \mathbf{step}(\langle b \rightarrow \bar{x} := \bar{e} \rangle)$, cf. Figure 1.

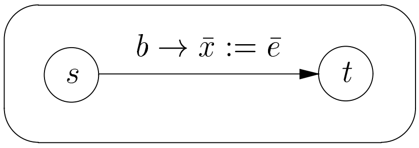

Figure 1: The transition system $T\,[\![\langle b \rightarrow \bar{x} := \bar{e} \rangle]\!]$.

- $T\,[\![\, S_1 \,;\, S_2 \,]\!] \stackrel{\text{def}}{=} T\,[\![S_1]\!] \;;\, T\,[\![S_2]\!]$, cf. Figure 2.

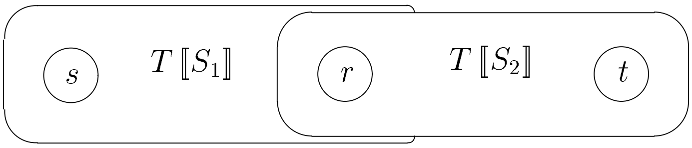

Figure 2: The transition system $T\,[\![S_1 \,;\, S_2]\!]$.

- $T\,[\![\, \mathbf{if} \;\; [\!]_{i=1}^{n} S_i \;\; \mathbf{fi} \,]\!] \stackrel{\text{def}}{=} \mathbf{if} \;\; [\!]_{i=1}^{n} T\,[\![S_i]\!] \;\; \mathbf{fi}$, cf. Figure 3.

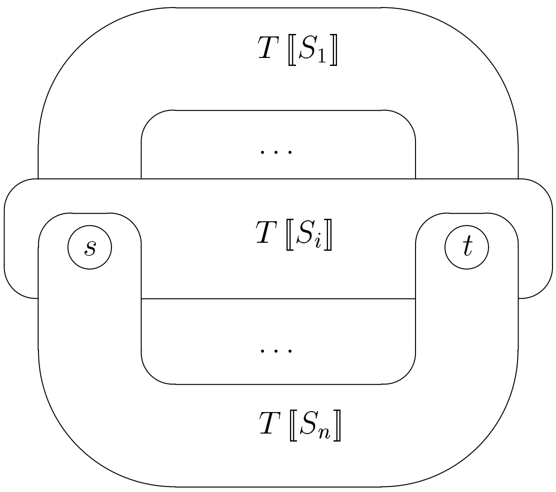

Figure 3: The transition system $T\,[\![\, \mathbf{if} \;\; [\!]_{i=1}^{n} S_i \;\; \mathbf{fi} \,]\!]$.

- $T[\![\mathbf{do}\ S_B\ [\!]\ S_E\ ;\mathbf{exit}\ \ \mathbf{od}]\!] \stackrel{\text{def}}{=} \mathbf{do}\ T[\![S_B]\!]\ [\!]\ T[\![S_E]\!]\ ;\mathbf{exit}\ \ \mathbf{od}$, cf. Figure 4. $\square$

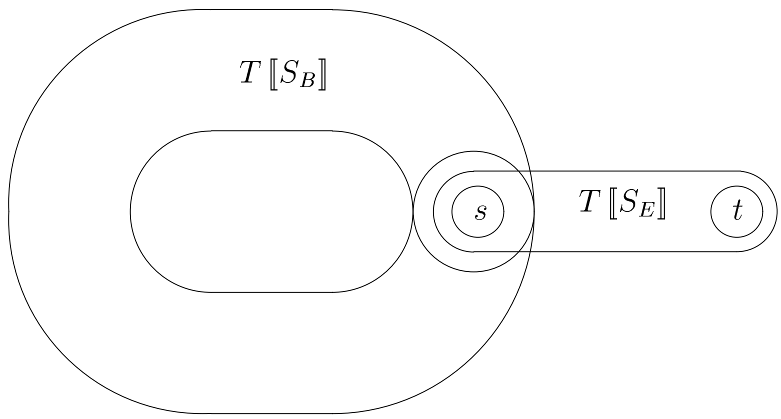

Figure 4: The transition system $T[\![\mathbf{do}\ S_B\ [\!]\ S_E\ ;\mathbf{exit}\ \ \mathbf{od}]\!]$.

Let us explain why the assumption of guardedness in the choice construct $\mathbf{if}\ \ [\!]_{i=1}^n S_i\ \ \mathbf{fi}$ and loop construct $\mathbf{do}\ \ S_B\ [\!]\ S_E\ ;\mathbf{exit}\ \ \mathbf{od}$ is essential. Consider for instance program $S_1 \stackrel{\text{def}}{=}$

$\mathbf{if}\ \ \mathbf{do}\ \langle x > 2 \rightarrow x := x - 2\rangle\ [\!]\ \langle x \leq 2 \rightarrow \mathbf{skip}\ \rangle\ ;\mathbf{exit}\ \ \mathbf{od}\ [\!]\ \langle \mathit{true} \rightarrow y := 2\rangle\ \mathbf{fi}$

with transition system $T[\![S_1]\!]$ given in Figure 5. Note that S_1 is not guarded because $\mathbf{do}\ \langle x > 2 \rightarrow x := x + 2\rangle\ [\!]\ \langle x \leq 2 \rightarrow \mathbf{skip}\ \rangle\ ;\mathbf{exit}\ \ \mathbf{od}$ is not guarded.

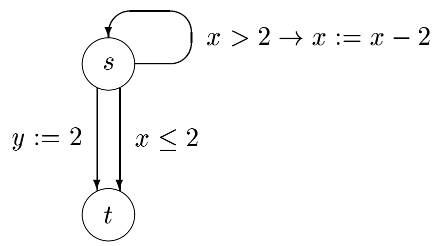

Figure 5: $T[\![S_1]\!]$.

Now one can easily show that there exist computations of S_1 which execute part of the loop

$$\mathbf{do}\ \langle x > 2 \rightarrow x := x + 2\rangle;\ [\!]\ \langle x \leq 2 \rightarrow \mathbf{skip}\ \rangle\ ;\mathbf{exit}\ \ \mathbf{od},$$

and then $\langle \mathit{true} \rightarrow y := 2\rangle$, without ever passing the guard $x \leq 2$. This is not consistent with the intuitive meaning of computations generated by the loop construct. For another example, consider the program

$$
\begin{array}{l}
S_2 \stackrel{\mathrm{def}}{=} \\
\quad \mathbf{if} \\
\quad\quad \mathbf{do}\ \langle x > 2 \rightarrow x := x - 2 \rangle\ [\!]\ \langle x \leq 2 \rightarrow \mathbf{skip}\ \rangle\ ;\ \mathbf{exit}\ \ \mathbf{od} \\
\quad [\!] \\
\quad\quad \mathbf{do}\ \langle y > 3 \rightarrow y := y - 1 \rangle\ [\!]\ \langle y \leq 3 \rightarrow \mathbf{skip}\ \rangle\ ;\ \mathbf{exit}\ \ \mathbf{od} \\
\quad \mathbf{fi}.
\end{array}
$$

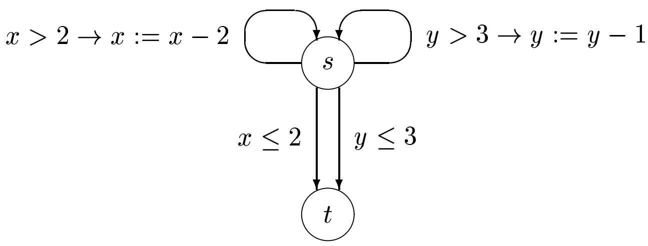

Figure 6: $T[\![S_2]\!]$.

The transition system $T[\![S_2]\!]$ is shown in Figure 6. As can be seen, this transition system allows for computations where execution of the body of one of the two loops alternates with execution of the body of the other loop, as long as both conditions $x > 2$ and $y > 3$ are satisfied. Of course this is not the intended meaning of the choice construct, where the choice which of the two loops will be executed is made immediately when the choice construct starts executing. This kind of anomaly can be excluded by imposing the guardedness condition. One can prove, that if S is guarded then for every transition diagram (L, T, s, t) representing $T[\![S]\!]$ one has $(l, a, s) \notin T$ for all labels l of L and for all actions a. That is, the start location s has no incoming transitions, and hence such anomalies, as illustrated above, cannot arise.

We can define an operational semantics $\mathcal{O}[\![P]\!]$ for arbitrary transition systems P as in Definition 11.1 of Session 11.

Definition 20.3 The initial-final state behaviour of transition system P represented by (L, T, s, t) is defined by:

$$\mathcal{O}[\![P]\!] \stackrel{\mathrm{def}}{=} \{(\sigma, \sigma') \mid \langle s; \sigma \rangle \rightarrow^* \langle t; \sigma' \rangle\},$$

where $\rightarrow^*$ denotes the reflexive transitive closure of the computation step relation $\rightarrow$ between configurations defined in Session 4. □

Indirectly, this also associates an operational semantics with GCL$^+$ programs:

Definition 20.4 (Operational semantics of GCL$^+$) For GCL$^+$ programs S we define: $\mathcal{O}[\![S]\!] \stackrel{\mathrm{def}}{=} \mathcal{O}[\![T[\![S]\!]]\!]$. □

(We have overloaded the $\mathcal{O}$ symbol here; from the context it is clear which function is meant). Notice this definition implies that we abstract from the internal states of a program. For instance $x := x+2$ and $x := x+1; x := x+1$ obtain the same semantics. Since we consider partial correctness in this chapter, only the initial and final state of a terminating computation are observable. Observe also that this semantics only records the effect of terminating computations. For example, $\mathcal{O}[\![\textbf{do}\ \ true \rightarrow \textbf{skip}\quad \textbf{od}]\!] = \emptyset$.

This definition of $\mathcal{O}[\![S]\!]$ is not a compositional one, as the definition of $\mathcal{O}[\![P]\!]$ is not in a compositional style.

References

[Bac78] R.J.R. Back. *On the Correctness of Refinement Steps in Program Development.* PhD thesis, Department of Computer Science, University of Helsinki, 1978. Report A-1978-4.

[Bac80] R.J.R. Back. Correctness preserving program refinements: Proof theory and applications. Mathematical Centre Tracts 131, Mathematisch Centrum, Amsterdam, 1980. Revised version of [Bac78].

[Bac90] R. Back. Refinement calculus, part II: Parallel and reactive programs. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Proceedings of the REX Workshop on Stepwise Refinement of Distributed Systems: models, formalisms, correctness, Mook, The Netherlands, May 29–June 2, 1989*, volume 430 of *LNCS*, pages 67–93. Springer-Verlag, 1990.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[Dij76] E.W. Dijkstra. *A discipline of programming.* Prentice Hall, 1976.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *CACM*, 12(10):576–580,583, 1969.