Presented at the Fifth Refinement Workshop, London,
8--10 January, 1992.

Formalising Dijkstra's Development Strategy within Stark's Formalism

Antonio Cau*
Institut für Informatik und Praktische Mathematik II
Preußerstr. 1-9
Christian-Albrechts-Universität zu Kiel
D-24105 Kiel, Germany

Ruurd Kuiper†
Department of Mathematics and Computing Science
Eindhoven University of Technology
5600 MB Eindhoven, The Netherlands

Willem-Paul de Roever‡
Institut für Informatik und Praktische Mathematik II
Preußerstr. 1-9
Christian-Albrechts-Universität zu Kiel
D-24105 Kiel, Germany

Abstract

Dijkstra introduced an enticing development strategy in a paper addressing the readers/ writers problem. This strategy is as follows: one starts with some "stupid" (in the sense that it allows undesirable computations) first try and then tries in subsequent steps to "refine" this stupid try into a better one by eliminating (some) undesirable computations. In a number of steps one strives to get a good (in the sense that it no longer contains undesirable computations) implementation for the problem. Unfortunately this strategy is not very formal. In this paper we try to make it more formal by using Stark's temporal logic based rely/guarantee formalism. We use this formalism in a special way in order to describe Dijkstra's development strategy: the part intended to describe the liveness condition is used for the more general purpose of disallowing the undesirable sequences.

1 Introduction

Current formal methods are far from solving the problems in software development. The simplest view of the formal paradigm is that one starts with a formal specification and subsequently develops a correct implementation which

*E-mail: ac@informatik.uni-kiel.d400.de
†E-mail: wsinruur@win.tue.nl
‡E-mail: wpr@informatik.uni-kiel.d400.de

is then proved to be correct. This view is too idealistic in a number of respects. First of all, most specifications of software are wrong and certainly most informal ones (unless they have been formally analyzed) contain inconsistencies [11]. Secondly, even a formal specification is produced (if at all) only after a number of iteration steps because writing a correct specification is a process whose difficulty is comparable with that of writing a correct program. This activity should therefore be structured, resulting in a number of increasingly less abstract layers with specifications which tend to increase in detail (and therefore become less readable [8]). Thirdly, even an incorrect program may describe a strategy whose specification by any other means is not as clear and has therefore at least *some* merits. This is especially the case with intricate algorithms such as those concerning specific strategies for solving the mutual exclusion problem. An interesting illustration of this third view is provided by E.W. Dijkstra's "Tutorial on the split binary semaphore" [2] in which he solves the readers/writers problem by subsequently improving an incorrect program till it is correct. If this master of style prefers to approximate and finally arrive at his correct solution using formally incorrect intermediate stages, one certainly expects that a formally correct development process for that paradigm is difficult to find! The strategy described in [2] is necessarily informal, reflecting the state of the art in 1979.

In the present paper we present a formal development strategy and its application to Dijkstra's example [2]. This formal strategy preserves the flavour of the informal strategy in that it formalises Dijkstra's argumentation in terms of incorrect approximations to a correct program and provides a formal criterion for recognising when a formally correct end product, the correct program, has finally been reached. We use Stark's formalism in order to achieve this. In this formalism a specification is separated in a safety (machine) part and a liveness (validity) part. It is this separation that enables us to handle incorrect approximations: the specific use of abstraction functions in Stark's formalism enables us to prove the correctness between machine parts, even in cases where incorrect sequences might prevent this in more rigid frameworks.

The structure of the paper is as follows: In Section 2 we introduce Stark's formalism and give some simplifications/improvements based on [3]. Furthermore we give an intuitive explanation of Stark's rely/guarantee rule for liveness properties. Stark's work was based on the rely/guarantee idea presented by Cliff Jones in [4]. We present in Section 3 the formal treatment of [2]. Section 4 contains a conclusion and mentions future work.

2 Stark's Formalism

2.1 Introduction

In this section we present Stark's formalism because papers [12, 13] are not easily accessible. We simplify his temporal logic; this simplification is based on that of [3]. Furthermore we give a more intuitive construction of the *events* of Stark's notion of composite machine and a more intuitive explanation of his rely/guarantee rule.

In Section 2.2 the notion of module is defined. In particular a distinction is made between abstract, composed and component modules. The idea is that an abstract module is implemented by a composed module which has component modules as components. Abstraction functions are defined and the notion of correct development step is defined, i.e., it is defined when a composed module implements an abstract one.

In order to relate these abstract black box notions to actual computations in Section 2.3 machines are introduced, a kind of automata. Stark's machine notion is a handy normal form to express safety properties. Lamport's notion of machine closure [1] can easily be applied to these machines. Liveness properties can be defined as global restrictions on the machine's behaviour. Stark makes a distinction between local properties and global ones, for instance, but not necessarily so, safety and liveness.

To obtain a more abstract temporal logic, doing away with the stuttering problem, Stark defines a dense linear time temporal logic. We adopt in Section 2.4 a slightly simplified/improved version of the logic as defined in [3]. The following picture illustrates the underlying model.

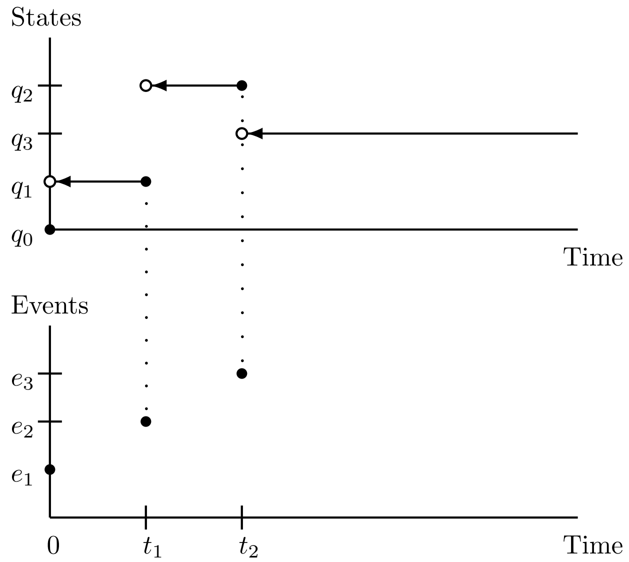


Figure 1: This picture illustrates the following: the initial state is q_0, on the occurrence of event e_1 the state changes in q_1. Between time 0 and time t_1 there are no interesting event occurrences, but only occurrences of the uninteresting stuttering event λ. So the state does not change until the next interesting event e_2 occurs.

A salient feature of the temporal logic is the "immediately after" state operator $'$, in a version which Lamport approves of according to [5].

In Section 2.5 machines and their allowed computations are related to correct development steps. The relation is expressed by verification conditions. In Section 2.6 we illustrate some of the notions of the previous sections with Lamport's soda machine example [7]. In Section 2.7 Stark's rely/guarantee notion for his proof rule is introduced. We also give an intuitive explanation of his proof rule and how he handles the problem of circular reasoning. In Section 2.8 we relate Stark's model to that of Lamport. In Section 2.9 we explain our special use of Stark's formalism in order to disallow undesirable sequences.

2.2 Modules and correct development steps

In [12] a method for specifying reactive systems is introduced. Such systems are assumed to be composed of one or more modules. A module is characterised by the specification pair $\langle E, B\rangle$ where E denotes its *interface* of possible events and B its *allowed behaviour*, as explained below.

An *event* is an observable instantaneous occurrence during the operation of a module, that can be generated by that module or its environment and that is of interest at the given level of abstraction. Also a λ_E-event which represents all uninteresting events (in Milners set up the τ event [10]) is distinguished.

The B-part of specification $\langle E, B\rangle$ characterises the allowed behaviour of the module. An *observation* x over interface E is a function from $[0, \infty)$ to E, such that $x(t) \neq \lambda$ for at most finitely many $t \in [0, \infty)$ in each bounded interval, which means that in a bounded interval only a finite number of interesting events can occur (this is the so called finite variability condition). Let $Obs(E)$ denote the set of all observations over E. Then the allowed behaviour B is a subset of $Obs(E)$. $Beh(E)$ denotes the set of all behaviours of interface E.

In Stark's view there are three kinds of modules. The first one is an *abstract* module. Such a module serves as a high level specification of a system. The second one is a *component* module which serves as a lower level specification of a system component. The third and last one is a *composite* module. This last module provides the link between the two levels.

To specify a system one needs one abstract module, one composite module and one or more component modules. An *interconnection* relates these modules with each other, i.e., it relates the interface of the composite module with the interface of the abstract module, and the interface of the composite module with each of the interfaces of the component modules.
An interconnection $\mathcal{I}$ is a pair $\langle \alpha, \langle \delta_i \rangle_{i \in I}\rangle$ where:

- α denotes a function from the interface E of the composite module to the interface A of the abstract module such that $\alpha(\lambda_E) = \lambda_A$ holds; α is called *abstraction* function.

- δ_i denotes a function from interface E of the composite module to interface F_i of the component module such that $\delta_i(\lambda_E) = \lambda_{F_i}$ holds; δ_i is called *decomposition* function.

The abstraction function α hides events from the composite machine that do not belong to the high level interface. The decomposition function δ_i hide events from the composite machine that do not belong to the component i. So intuitively the requirement about both α and the δ_i's is that uninteresting events of the composite module are not turned into interesting ones of the abstract or component modules.

The definition of interconnection can easily be extended to behaviour of the mentioned modules. When $\mathcal{I}$ is an interconnection between the interfaces of the modules, $\mathcal{I}^*$ denotes the corresponding interconnection between the behaviours of the modules.
If $\mathcal{I}$ is a pair $\langle \alpha, \langle \delta_i \rangle_{i \in I} \rangle$ then $\mathcal{I}^*$ is the pair $\langle \alpha^*, \langle \delta_i^* \rangle_{i \in I} \rangle$ where :

- α^* denotes a function from the set $Beh(E)$ of all possible behaviours of the composite module to the set $Beh(A)$ of all possible behaviours of the abstract module. If $B_E \in Beh(E)$ then $\alpha^*(B_E) \stackrel{\text{def}}{=} \{\alpha \circ x \mid x \in B_E\}$ is obtained by elementwise composition of α.

- δ_i^* denotes a function from the set $Beh(E)$ of all possible behaviours of the composite module to the set $Beh(F_i)$ of all possible behaviours of the component module. If $B_E \in Beh(E)$ then $\delta_i^*(B_E) \stackrel{\text{def}}{=} \{\delta_i \circ x \mid x \in B_E\}$.

It is the composite module that actually defines the composition of the component modules. In the examples we define this composite module in such a way that it reflects the parallel composition of the component modules. A *development step* is defined as a triple $\langle \mathcal{I}^*, S_A, \langle S_i \rangle_{i \in I} \rangle$ where

- $\mathcal{I}^*$ is the interconnection (between behaviours),

- S_A is the specification of the abstract module : $\langle A, B_A \rangle$
($B_A \in Beh(A)$ denotes the set of allowed behaviours of the abstract module), and

- S_i is the specification of component module i : $\langle F_i, B_i \rangle$
($B_i \in Beh(F_i)$ denotes the set of allowed behaviours of component module i).

The inverse image operator induced by the decomposition functions of an interconnection expresses the operation of composing a collection of component modules to produce the corresponding behaviour of the composite module.
Hence, the composition operator associated with $\langle \delta_i^* \rangle_{i \in I}$ is the composition function $\langle \delta_i^* \rangle_{i \in I}^{-1}$, mapping the vector $\langle B_i \rangle_{i \in I}$ of allowed behaviours of the component modules to a composite module's behaviour $\langle \delta_i^* \rangle_{i \in I}^{-1}(\langle B_i \rangle_{i \in I}) \in Beh(E)$ under the definition:

$$\langle \delta_i^* \rangle_{i \in I}^{-1}(\langle B_i \rangle_{i \in I}) = \{x_E \in Obs(E) \mid \delta_i \circ x_E \in B_i \text{ for all } i \in I\} = \bigcap_{i \in I} \delta_i^{*-1}(B_i)$$

A development step is correct if the behaviour obtained by abstracting away the internal behaviour (of the composed component modules) is also a behaviour of the abstract module; i.e., if the following holds:

$$\alpha^* \circ (\langle \delta_i^* \rangle_{i \in I}^{-1})(\langle B_i \rangle_{i \in I}) \subseteq B_A$$

2.3 Machines

Until now we have specified the allowed behaviour of a module by a set of observations. We now introduce a state-transition formalism to generate this set. In this state-transition formalism, we imagine that – at any instant of time – a module can be thought of as being in a *state*. Associated with each state is a collection of events that can occur in that state, and a description of the state change that results from the occurrence of each of those events. Thus a state-transition specification describes the desired functioning of a module in terms of a machine that generates an observation as it executes.

One can divide the properties that can be specified by the state-transition technique in two classes. The first class consists of the so called *local (safety) properties*, which describe how an event causes a state to transform to the next state. The second class consists of the so called *global properties*, which describe the relationship of events and states that cannot be directly described in terms of state-transition relations.

The local properties are specified by the above mentioned machine and the global properties are specified by defining a set of *validity conditions* on computations of that machine. The set of computations that satisfy the validity conditions is called the *set of valid computations*. The intersection of this set with the set of computations that are generated by the machine, describe the allowed behaviour of the corresponding module.

The machine M that specifies the local properties of a module is defined as follows:
$M = (E_M, Q_M, IQ_M, TR_M)$ where:

- E_M : is the interface of M; events labeled with a$\downarrow$ are input events, events labeled with a$\uparrow$ are output events and events without an arrow are internal events,

- Q_M : is the set of states of M; a state is a function from the set of observable variables Var to the set of values Val i.e. $Q_M : Var \to Val$,

- IQ_M : a non-empty subset of Q_M, the set of initial states,

- TR_M : the state-transition relation, $TR_M \subseteq Q_M \times E_M \times Q_M$, such that for all $q \in Q_M$ the stuttering step $\langle q, \lambda_{E_M}, q\rangle \in TR_M$. Furthermore M is input-cooperative (if an input comes "at the wrong moment" it should be mapped to *error*, i.e., TR_M is total for input events).

The next example illustrates how such a specification of a machine M may look like.

Example

$M = (E_M, Q_M, IQ_M, TR_M)$ where:

1. **Events:**
   E_M : $\{d_0, d_1, \lambda_d\}$

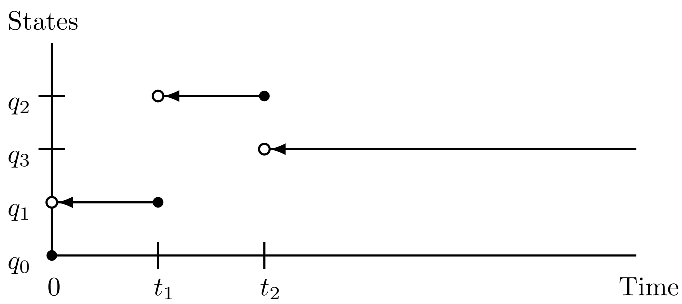
2. **States:**
   $Q_M : \{u\} \to \{0, 1, 2\}$

3. **Initial States:**
   IQ_M: $\{q \in Q_M : q(u) = 0\}$

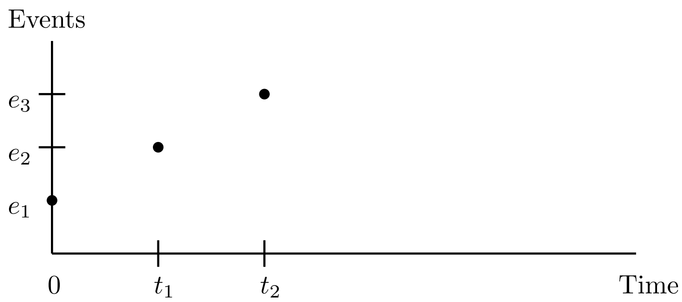
4. **Transitions:**
   TR_M:
   $$\{\langle q, e, r\rangle \in Q_M \times E_M \times Q_M :$$
   (a) $(q(u) = 0 \wedge e = d_0 \wedge r(u) = 1) \vee$
   (b) $(q(u) = 1 \wedge e = d_1 \wedge r(u) = 2) \vee$
   (c) $(e = \lambda_d \wedge r(u) = q(u))\}$

   <u>end example</u>

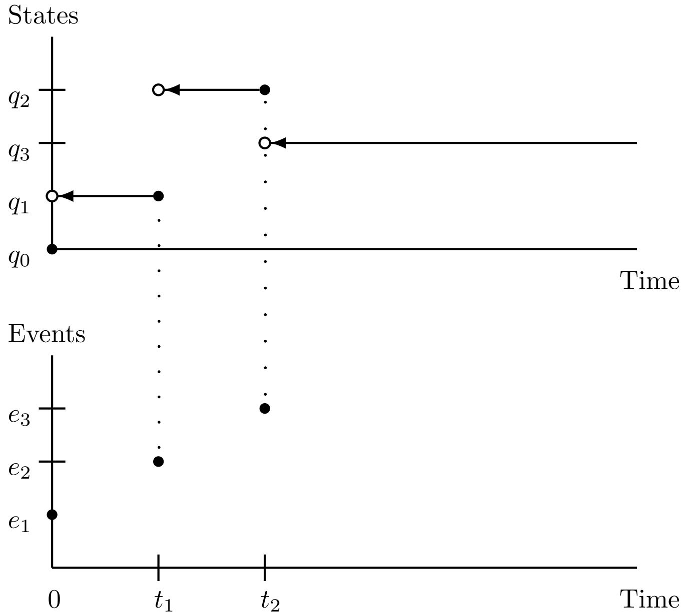
A *state function* over a set of states Q is a function $f : [0, \infty) \to Q$ such that for all $t \in [0, \infty)$, there exists $\varepsilon_t > 0$ such that f is constant on intervals $(t - \varepsilon_t, t] \cap [0, \infty)$ and $(t, t + \varepsilon_t]$. We write $f(t^{\leftarrow\bullet})$ for the value of the state just before and at time t (the first interval) and write $f(t^{\circ\rightarrow})$ for the value of f just after time t (the second interval). This is illustrated in the next picture where the state just before and at time t_1 equals q_1 and the state just after time t_1 equals q_2.



A *history* over an interface E and state set Q is a pair $X = \langle Obs_X, State_X\rangle$, where Obs_X is an observation over E (a function from $[0, \infty)$ to E). An example of such an observation is illustrated in the next picture. For example at time t_1 event e_2 occurs.



And where $State_X$ is a state function over Q as illustrated in the first picture. These two notions will be related by the notion of computation of a machine M. But first we need the notion of step at t in X.
Let $Hist(E,Q)$ denote the set of all histories over interface E and state set Q. If $X \in Hist(E,Q)$ and $t \in [0,\infty)$, then define the *step* occurring at time t in X by:
$Step_X(t) = \langle State_X(t^{\leftarrow\bullet}), Obs_X(t), State_X(t^{\circ\!\!\rightarrow}) \rangle$. An example of such a step is illustrated in the next picture whichs combines the previous two pictures. The step at time t_1 is then $\langle q_1, e_2, q_2 \rangle$.



A *computation of a machine* M is a history $X \in Hist(E_M, Q_M)$ such that:

- $State_X(0) \in IQ_M$.
- $Step_X(t) \in TR_M$ for all $t \in [0, \infty)$.

Let $Comp(M)$ denote the set of all computations of M.
Let $Reachable_M$ denote the set of reachable states of M, i.e. all the states that can occur in a computation.

If V is a set of computations of M, then define $Obs(V)$ –the set of all observations generated by V – by $Obs(V) = \{Obs_X : X \in V\}$.

2.4 Stark's dense linear time logic DTL

As we have seen above the global properties are described by a set of validity conditions. Stark uses temporal logic to describe these validity conditions. Our modified temporal logic DTL looks like the one of Stark and is defined as follows:
Let F denote the set of freeze variables then a *history model* is a tuple $\langle\theta, h\rangle$, where $\theta \in F \to Val$ is an assignment to freeze variables and h a history (pair $\langle Obs_h, State_h\rangle$). Let $h^{(\tau)}$ denote the history $\lambda t.h(t + \tau)$.

Syntax

variables elements of Var
values of variables elements of Val
mapping operator $\mapsto$
state mapping from Var to Val, i.e. when for instance $Var = \{x\}$ and

$Val = \{0, 1, 2\}$ then $[x \mapsto 0]$ denotes a state
freeze variables elements of F; $F \cap Var = \emptyset$
events elements of E_M
special symbols **e** and **st**
event term $\mathbf{e} = f$ where f denotes an element of E_M
state terms $\mathbf{st} = x$ and $\mathbf{st}' = x$ where x denotes a state and $'$ is a **temporal** operator
terms can be event terms, state terms, freeze variables or function symbols
quantification over freeze variables $\forall, \exists$
formulae built from terms, relation symbols, boolean connectives, quantification and **temporal** operators $\Box$ and $\Diamond$

Semantics

Before we give the semantics of DTL we give the definition of a variant of a state q: $(q \mid x : v)$ which is defined as follows:
$(q \mid x : v)(y) = v$ for $y = x$ and $(q \mid x : v)(y) = q(y)$ for $y \neq x$.
For all freeze variables v, $v(\langle \theta, h \rangle) = \theta(v)$.
For all variables $v \in Var$, $v(\langle \theta, h \rangle) = State_h(0)(v)$.
For $[v \mapsto n]$, where $v \in Var$ and $n \in Val$, $[v \mapsto n](\langle \theta, h \rangle) = State_h(0) \mid v : n$
For **e** , $\mathbf{e}(\langle \theta, h \rangle) = Obs_h(0)$.
For **st**, $\mathbf{st}(\langle \theta, h \rangle) = State_h(0)$.
For **st′**, $\mathbf{st}'(\langle \theta, h \rangle) = State_h(0^{\circ\!\!\to})$.
As usual.
For function f with interpretation $\overline{f}$,
$f(t_1, ..., t_n)(\langle \theta, h \rangle) = \overline{f}(t_1(\langle \theta, h \rangle), ..., t_n(\langle \theta, h \rangle))$.
$\langle \theta, h \rangle \models R(t_1, ..., t_n)$ if $\overline{R}$ is the interpretation of R and $\overline{R}(t_1(\langle \theta, h \rangle), ..., t_n(\langle \theta, h \rangle))$ holds;
$\langle \theta, h \rangle \models \neg\varphi$ if $\langle \theta, h \rangle \not\models \varphi$;
$\langle \theta, h \rangle \models \varphi \rightarrow \psi$ if $\langle \theta, h \rangle \models \neg\varphi$ or $\langle \theta, h \rangle \models \psi$;
$\langle \theta, h \rangle \models \exists x.\varphi$ if there exists an assignment θ' differing from θ only in the value assigned to freeze variable x such that $\langle \theta', h \rangle \models \varphi$;
$\langle \theta, h \rangle \models \Diamond\varphi$; if there exists an $t \in [0, \infty)$ such that $\langle \theta, h^{(t)} \rangle \models \varphi$;
$\langle \theta, h \rangle \models \Box\varphi$; if for all $t \in [0, \infty)$ $\langle \theta, h^{(t)} \rangle \models \varphi$;

The initial states and the transition relation can also be expressed as a DTL formula, as illustrated in the next example. Note that although we use the same names IQ_M and TR_M as in the previous example, this is in fact not correct because in the next example these are actual DTL formulae. When we refer to these names we mean from now on the DTL formulae.

Example

Same machine M as above:

1. **Events:**
   $E_M : \{d_0, d_1, \lambda_d\}$

2. **States:**
   $Q_M : \{u\} \rightarrow \{0, 1, 2\}$

3. **Initial States:**
   $IQ_M \equiv \mathbf{st} = [u \mapsto 0]$

4. **Transitions:**
   $$\begin{aligned} TR_M \equiv\ & (\mathbf{st} = [u \mapsto 0] \wedge \mathbf{e} = d_0 \wedge \mathbf{st}' = [u \mapsto 1]) \vee \\ & (\mathbf{st} = [u \mapsto 1] \wedge \mathbf{e} = d_1 \wedge \mathbf{st}' = [u \mapsto 2]) \vee \\ & (\mathbf{e} = \lambda_d \wedge \mathbf{st}' = \mathbf{st}) \end{aligned}$$

   end example

In the above example we used expressions like $\mathbf{st} = [u \mapsto 0]$; to increase readability we use the abbreviation $\mathbf{st}(u) = 0$ instead of the previous one.

The *enabling condition* of an event in machine M denoted by $Enabled_M(e)$ is that condition that enables the generation of that event in M. For example, $Enabled_M(d_0)$ of the previous example is condition $\mathbf{st}(u) = 0$.

In order to describe situations where an old state is updated we use the variant construct: $\mathbf{st} \mid x : v$ defined above. Furthermore we do not mention the λ-transition anymore because this transition is the same for all machines.

The local properties of a module Z can now be expressed as formula $IQ_Z \wedge \Box TR_Z$. Thus Comp(M_Z)=$\{X \in Hist(E_Z, Q_Z) \mid X \models IQ_Z \wedge \Box TR_Z\}$. The liveness properties can now be added, expressed by some extra DTL formula V_Z, the *validity condition*. The complete behaviour of module Z is the following set of histories:

$$\{X \in \text{Comp}(M_Z) \mid X \models V_Z\},$$

and is described by formula $IQ_Z \wedge \Box TR_Z \wedge V_Z$.

2.5 Machines, allowed computations, and correct development steps

As we have seen above, there are several kinds of machines -abstract, component and composite ones- and they all have a set of allowed computations. If we have an abstract machine M_A, described by temporal formula $IQ_A \wedge TR_A$, and component machines M_i, described by temporal formula $IQ_i \wedge TR_i$, and if we have furthermore an interconnection $\mathcal{I} = \langle \alpha, \langle \delta_i \rangle_{i \in I} \rangle$ that links both kinds of machine then we can construct the composite machine M_c as follows:

- The interface E_c is the same as the interface of the interconnection.
- The set of states $Q_c = Q_A \times \prod_{i \in I} Q_i$, i.e, the product of the set of states of the abstract machine with the product of the sets of states of all the component machines.

- The set of initial states of M_c we also want to describe by a temporal formula. A first try would be the following temporal formula $IQ_A \wedge \bigwedge_{i \in I} IQ_i$ but this formula consists of a part that describe the initial states of M_A and a part that describe the initial states of the M_i's. The formula must however describe the initial states of M_c. But fortunately the set of states of M_c is defined as a Cartesian product of the set of states of M_A and M_i. So if we replace every state term $\mathbf{st} = x$ in IQ_A by $\pi^A(\mathbf{st}) = x$ where π^A is just the ordinary projection function from Q_c to Q_A, then this last formula expresses the same thing as IQ_A but now in terms of the states of M_c. This replacement is denoted by $[IQ_A]_{\mathbf{Atoc}}$.

  The same thing can be done for the temporal formulas IQ_i: state term $\mathbf{st} = x$ is replaced by $\pi^i(\mathbf{st}) = x$ where π^i is just the ordinary projection function from Q_c to Q_A. This replacement is denoted by $[IQ_i]_{\mathbf{itoc}}$.

  So the set of initial states of M_c can be expressed by following temporal formula $IQ_c \stackrel{\text{def}}{=} [IQ_A]_{\mathbf{Atoc}} \wedge \bigwedge [IQ_i]_{\mathbf{itoc}}$.

- For describing the state-transition relation we have the same problem but now for the states and events. But fortunately we have the definition of α and δ_i's which we can use to transform event terms in TR_A and TR_i into event terms of TR_c. Event term $\mathbf{e} = d$ in TR_A is transformed into $\alpha(\mathbf{e}) = d$ and event term $\mathbf{e} = f$ in TR_i is transformed into $\delta(\mathbf{e}) = f$. Let $[f]_{\mathbf{Atoc}}$ denote the transformation of both event and state terms of a formula f in the temporal framework of the abstract machine into the temporal framework of the composite machine and let $[g]_{\mathbf{itoc}}$ denote the transformation of both event and state terms of a formula g in the temporal framework of a component machine i. Then the state-transition relation of the composite machine can now be expressed as following temporal formula $TR_c \stackrel{\text{def}}{=} \Box([TR_A]_{\mathbf{Atoc}} \wedge \bigwedge_{i \in I} [TR_i]_{\mathbf{itoc}})$.

We use the correctness definition given before in the following form:

$\langle \delta_i^* \rangle_{i \in I}^{-1}(\langle B_i \rangle_{i \in I}) \subseteq \alpha^{-1}(B_A)$.

In the present formalism, this translates to

$\langle \delta_i^* \rangle_{i \in I}^{-1}(\langle \{X \in Comp(M_i) : X \models V_i\} \rangle_{i \in I}) \subseteq \alpha^{-1}(\{X \in Comp(M_A) : X \models V_A\})$.

And this can be expressed as the following temporal formula:

$\bigwedge_{i \in I} [IQ_i \wedge \Box TR_i \wedge V_i]_{\mathbf{itoc}} \rightarrow [IQ_A \wedge \Box TR_A \wedge V_A]_{\mathbf{Atoc}}$

Due to the separation of the allowed behaviour into a machine and a validity part we can split this verification condition into two verification conditions. One applying to machines and one applying to validity conditions[1]:

[1] Observe that this split can only be done when V_i and V_A concern pure liveness properties cfr. [1]. In case $V = \Box S_0 \wedge V'$ for a validity condition V, where $\Box S_0$ is the safety part and V' the pure liveness part (see [1] for this terminology) the transition relation TR of the machine in question must be described by $TR' \stackrel{\text{def}}{=} TR \wedge S_0$ and the validity part by V'.

- *maximality* : any event that can be generated by the system of component machines can also be performed by the abstract machine.
- *validity* : any allowed computation of each component machine corresponds with an allowed computation of the abstract machine.

More formally:

- *maximality* :
  $Comp(M_c) \models \forall e \in E_c.(Reachable_c \wedge \bigwedge_{i\in I}[Enabled_i(e)]_{\mathbf{itoc}})$
  $\rightarrow [Enabled_A(e)]_{\mathbf{Atoc}}$
  where $Reachable_c$ is a condition that checks if a state of the composite machine is reachable, $Enabled_i(e)$ is the enabling condition of the event of machine i corresponding to event e, and $Enabled_A(e)$ the enabling condition of the event of the abstract machine corresponding event to event e.
- *validity* :
  $Comp(M_c) \models (\bigwedge_{i\in I}[V_i]_{\mathbf{itoc}}) \rightarrow [V_A]_{\mathbf{Atoc}}$
  where V_i is the validity condition of module i, and V_A is the validity condition of the abstract module.

Given the construction of the composite machine, maximality implies, intuitively, that all interleavings, even unfair ones, of the component machines should, after abstraction, be allowed by the abstract machine. Validity means, intuitively, that only those sequences should be allowed as complete behaviours that also satisfy some progress properties.

To prove these two conditions it is necessary to find an implementation invariant that, firstly, describes the reachable states of the composite machine in order to prove the maximality condition and, secondly, is such that it is of help in the proof of the validity condition.

The proof of the maximality condition is intuitively done as follows: one checks if for all events of the composite machine the maximality condition holds. For the proof of the validity condition Stark uses his rely/guarantee rule because the V-formulae can be written in rely/guarantee form. This rule solves the circular reasoning problem in another way than [4, 15, 9], see Section 2.7 for details.

2.6 Specification of Lamport's soda machine

In the next example, the soda machine example [7], we illustrate some of the above notions – particularly that of composite machine. The soda machine is a system in which the user deposits either a half dollar or two quarters and the machine in return dispenses a can of soda.

Example

Given two specifications of a soda-machine, show that one specification implements (i.e. refines) the other one.
The high level specification of the soda-machine :

Initially the user either deposits a quarter or a half dollar. If he deposits a quarter then the next coin can only be a quarter. If he next deposits a half dollar the machine enters the error state; if he deposits a quarter the machine dispenses a can of soda. A can of soda is also returned when he deposits a half dollar initially. If the user deposits another coin before the machine has dispensed a can of soda then the machine will also enter the error state. This informal specification is illustrated in figure 2 and written down formally in Stark's formalism as S_H:

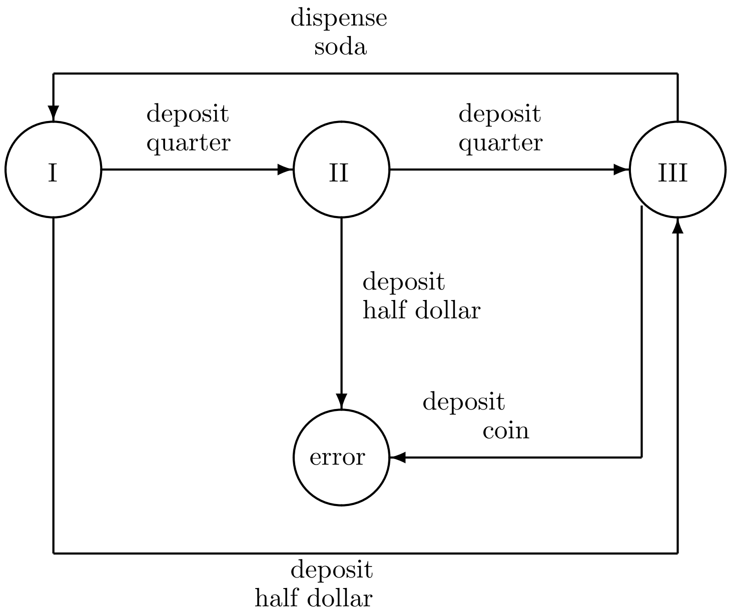


Figure 2:

1. **Events:**
   $E_H = \{\text{de.qu}\downarrow, \text{de.hd}\downarrow, \text{di.so}\uparrow, \lambda_H\}$
   - de.qu$\downarrow$: the depositing of a quarter by the environment.
   - de.hd$\downarrow$: the depositing of a half dollar by the environment.
   - di.so$\uparrow$: the dispensing of a can of soda by M_H.
2. **States**
   $Q_H : ms \rightarrow \{I, II, III, error\}$
   $ms = I$: M_H is in state I.
3. Initial States:
   $IQ_H \equiv \mathbf{st}(ms) = I$
4. **Transitions:**
   $TR_H \equiv$

- $\mathbf{e} = \text{de.qu}{\downarrow} \wedge ((\mathbf{st}(ms) = I \wedge \mathbf{st}' = \mathbf{st} \mid ms : II) \vee$
  $(\mathbf{st}(ms) = II \wedge \mathbf{st}' = \mathbf{st} \mid ms : III) \vee$
  $((\mathbf{st}(ms) \neq I \vee \mathbf{st}(ms) \neq II) \wedge \mathbf{st}' = \mathbf{st} \mid ms : error))$
  If the environment deposits a quarter and M_H is in state I then it enters state II. If the environment deposits a quarter and M_H is in state II then M_H enters state III. If the environment deposits a quarter and M_H is in state III or the *error* state then M_H enters the *error* state. This latter possibility we must explicitly allow, because an machine M must be input-cooperative.
- $\mathbf{e} = \text{de.hd}{\downarrow} \wedge ((\mathbf{st}(ms) = I \wedge \mathbf{st}' = \mathbf{st} \mid ms : III) \vee$
  $(\mathbf{st}(ms) \neq I \wedge \mathbf{st}' = \mathbf{st} \mid ms : error))$
  If the environment deposits a half dollar and M_H is in state I then M_H enters state III. In all other states M_H enters the *error* state.
- $\mathbf{e} = \text{di.so}{\uparrow} \wedge \mathbf{st}(ms) = III \wedge \mathbf{st}' = \mathbf{st} \mid ms : I$
  M_H dispenses a can of soda when it is in state III and then enters state I.

5. **Validity Condition:**
   To make the example not too complicated we leave the validity condition out of the specification, but it should specify intuitively that if the user deposits a half dollar or 2 quarters then eventually the machine dispenses a can of soda.

The lower level specification :
The lower level specification is written in some programming language:

var x: { 0, 25, 50 };
    $y : \{25, 50\}$;
beginloop
    $\alpha : \langle x := 0 \rangle$;
    β : **while** $\langle x < 50 \rangle$
            do $\gamma : \langle y := \textit{deposit\_coin}$ **only if** $x + y_{new} \leq 50 \rangle$ **else raise error**
                $\delta : \langle x := x + y \rangle$
            od;
    $\epsilon : \langle \textit{dispense\_soda} \rangle$
end loop
$error : \langle errorhandling \rangle$

The only construct that should need explanation is:
$\langle y := \textit{deposit\_coin}$ **only if** $x + y_{new} \leq 50 \rangle$ **else raise error**.
The meaning is as follows: y is assigned the value of a coin only if this value plus the current value of x is less or equal 50, otherwise it will raise an error, i.e. enter the error state.
The above program is in Stark's formalism described as specification S_L:

1. **Events:**
   $E_L = \{e1, e2, e3(v){\downarrow}, e4, e5{\uparrow}, \lambda_L : v = \{25, 50\}\}$

2. **States:**
Q_L : $(pc \to \{\alpha, \beta, \delta, \gamma, \epsilon, error\}) \times (x \to \{0, 25, 50\}) \times (y \to \{25, 50\})$

3. **Initial States**
$IQ_L \equiv \mathbf{st}(pc) = \alpha$

4. **Transitions:**
$TR_L \equiv$

   - $\mathbf{e} = e1 \wedge \mathbf{st}(pc) = \alpha \wedge \mathbf{st'} = \mathbf{st} \mid pc, x : \beta, 0$
   M_L performs an internal event to make x equal 0.
   - $\mathbf{e} = e2 \wedge \mathbf{st}(pc) = \beta \wedge ((\mathbf{st}(x) < 50 \wedge \mathbf{st'} = \mathbf{st} \mid pc : \gamma) \vee$
   $(\mathbf{st}(x) \geq 50 \wedge \mathbf{st'} = \mathbf{st} \mid pc : \epsilon))$
   M_L performs an internal checking event.
   - $\mathbf{e} = e3(v){\downarrow} \wedge((\mathbf{st}(pc) = \gamma \wedge v + \mathbf{st}(x) \leq 50 \wedge \mathbf{st'} = \mathbf{st} \mid pc, y : \delta, v) \vee$
   $((\mathbf{st}(pc) \neq \gamma \vee v + \mathbf{st}(x) > 50) \wedge \mathbf{st'} = \mathbf{st} \mid pc : error))$
   The environment deposits a quarter or a half dollar: this may only happen if pc and x have certain values. If pc and x do not have these values M_L enters the *error* state. What coin is deposited is "remembered" by y.
   - $\mathbf{e} = e4 \wedge \mathbf{st}(pc) = \delta \wedge \mathbf{st'} = \mathbf{st} \mid pc, x : \beta, \mathbf{st}(x) + \mathbf{st}(y)$
   M_L performs an internal adding event.
   - $\mathbf{e} = e5{\uparrow} \wedge \mathbf{st}(pc) = \epsilon \wedge \mathbf{st'} = \mathbf{st} \mid pc : \alpha$
   The machine dispenses a can of soda.

5. **Validity Condition:**
To make the example not too complicated we leave the validity condition out of the specification.

We must find a composite machine M_C and translations α and δ such that "$\alpha(M_C)$ gives M_H and $\delta(M_C)$ gives M_L". First find translations α and δ. From the definition of the abstraction and decomposition operator we know that α is a function from E_C (interface of composite machine) to E_H and δ is a function from E_C to E_L. We can construct E_C by means of a Cartesian product out of E_H and E_L using some intuition. (This construction is not from [12, 13] but in our opinion more clearly follows the intuition.)

If M_L generates $e1$ (an internal event) then M_H must generate a λ_H event thus $\langle \lambda_H; e1 \rangle$ is an event of M_C. We can do this for all the events of M_H and M_L. E_C is then as follows:
$E_C = \{\langle \lambda_H, e1 \rangle, \langle \lambda_H, e2 \rangle, \langle \text{de.qu}{\downarrow}, e3(25){\downarrow} \rangle, \langle \text{de.hd}{\downarrow}, e3(50){\downarrow} \rangle,$
$\langle \lambda_H, e4 \rangle, \langle \text{di.so}{\uparrow}, e5{\uparrow} \rangle, \langle \lambda_H, \lambda_L \rangle\}$
From E_C we now can get α and δ:
Let $\langle p, q \rangle \in E_C$ then $\alpha(\langle p, q \rangle) = p$ and $\delta(\langle p, q \rangle) = q$. The construction of M_C is now easy, see Section 2.5.

In order to check that M_L implements M_H we must check:

$$\forall e \in E_C.[Enabled_L(e)]_{L\mathbf{toc}} \to [Enabled_H(e)]_{H\mathbf{toc}}$$

This formula holds, for instance in case of $e = \langle \lambda_H, e1 \rangle$:
$[Enabled_L(e)]_{\mathbf{Ltoc}} \equiv \pi^L(\mathbf{st})(pc) = \alpha$ and $[Enabled_H(e)]_{\mathbf{Ltoc}} \equiv true$ so the formula holds for case $e = \langle \lambda_H, e1 \rangle$.

end example

2.7 Stark's rely/guarantee proof rule

It is Stark's intention to prove the validity part with a proof rule. To use this proof rule it is necessary that the set of allowed computations is in a special form. This form, which is called the *rely/guarantee* form is based on Cliff Jones' original idea of including, in the rely-condition, assumptions about the other components [4]. In Stark's formalism this form is as follows: $\{X \in \text{Comp}(M) : X \models R \to G\}$, i.e. the DTL logic formula V is written as $R \to G$. The R(ely)-part of this formula expresses how the module being specified relies on what its environment provides. The G(uarantee)-part of this formula expresses what the module then guarantees to provide. We now give an intuitive explanation of this proof rule.

As seen above we must show that any allowed computation of each component machine corresponds with an allowed computation of the abstract machine. The sets of allowed computations of component machine M_i and abstract machine M_A are respectively expressed as $\{X \in \text{Comp}(M_i) : X \models R_i \to G_i\}$ and $\{X \in \text{Comp}(M_A) : X \models R_A \to G_A\}$. In order to compare these sets with each other we must transform these sets into sets that specify the allowed computations of the composite machine. This means that the first set is transformed into $\{X \in \text{Comp}(M_c) : X \models [R_i]_{\mathbf{itoc}} \to [G_i]_{\mathbf{itoc}}\}$ and the second one into $\{X \in \text{Comp}(M_c) : X \models [R_A]_{\mathbf{Atoc}} \to [G_A]_{\mathbf{Atoc}}\}$. In order to facilitate the explanation of the proof rule we write $\mathcal{P} \models R_A \to G_A$ instead of $\{X \in \text{Comp}(M_c) : X \models [R_A]_{\mathbf{Atoc}} \to [G_A]_{\mathbf{Atoc}}\}$ and $\mathcal{P} \models R_i \to G_i$ instead of $\{X \in \text{Comp}(M_c) : X \models [R_i]_{\mathbf{itoc}} \to [G_i]_{\mathbf{itoc}}\}$.

There must exist some relationship between the R_A, G_A, R_i and G_i mentioned above. This relationship is as follows:

- Relationship between G_i and G_A:
  If the conjunction of the G_i's holds then G_A holds.
  Formally: $\mathcal{P} \models \bigwedge_i G_i \to G_A$.

- Relationship between R_i and R_A:
  If R_A holds then it is impossible to infer that the conjunction of the R_i's holds because the R_A only says something about the external relationship and the R_i says something about the internal relationship too. Thus R_A is not enough. We also need a condition to infer that the internal relationship holds. This condition is the conjunction of the G_i because this guarantees the internal relationship.
  Formally: $\mathcal{P} \models [R_A \wedge \bigwedge_{j \neq i} G_j] \to R_i$

This leads to the following rule to infer $\mathcal{P} \models R_A \to G_A$ from the $\mathcal{P} \models R_i \to G_i$'s: (Note: this rule is nearly the same as in [13].)

$$\frac{\mathcal{P} \models R_i \to G_i, \mathcal{P} \models [R_A \wedge \bigwedge_{j \neq i} G_j] \to R_i, \mathcal{P} \models \bigwedge_i G_i \to G_A}{\mathcal{P} \models R_A \to G_A}$$

Regrettably, this simple rule is not sound in our set-up (a similar rule is, however, sound in the setting of [4, 15, 9] because there they do not define a R/G condition to hold for a component by straightforward implication, as above, but by a more involved definition reflecting induction on the communication trace).

The reason is that one can get a cycle of proof obligations, as can be seen from the following example.

Example

Suppose there are two component machines. Suppose we have proven the following proof obligations:

1. $\mathcal{P} \models R_1 \to G_1$
2. $\mathcal{P} \models R_2 \to G_2$
3. $\mathcal{P} \models [R_A \wedge G_2] \to R_1$
4. $\mathcal{P} \models [R_A \wedge G_1] \to R_2$
5. $\mathcal{P} \models [G_1 \wedge G_2] \to G_A$

We want to infer $\mathcal{P} \models R_A \to G_A$ from 1–5.
Assume R_A holds.
Then we must prove that G_A holds.
Working our way backwards, we conclude: $G_1 \wedge G_2$ should hold.
Similarly, from 1 and 2 we conclude: $R_1 \wedge R_2$ should hold.
From 3 and 4 we conclude: $[R_A \wedge G_2] \wedge [R_A \wedge G_1]$ should hold.
Thus, $G_1 \wedge G_2$ should hold.
So we get the cycle $G_1 \wedge G_2 \to R_1 \wedge R_2 \to G_1 \wedge G_2$.

end example

The idea is now, that the proof obligations in the cycle can be trivially fulfilled by choosing false for all assertions involved. It is than also possible to choose R_A to be true and G_A to be false. $\mathcal{P} \models R_A \to G_A$ is than derivable, and equivalent to true implies false! Hence the rule is unsound. Stark eventually solves this problem by imposing a condition that rules out cycles. To avoid making the rule seriously incomplete by imposing such a condition, first an adaptation to reflect the dependencies between components more precisely needs to be made.

In the rule as given above we can not see the split between the internal and external relationship of a component. Therefore Stark introduces $G_{i,j}$, $G_{i,A}$, $G_{A,i}$, $R_{i,j}$, $R_{A,i}$ and $R_{i,A}$ to make this split explicit. (In the following, "environment" means the environment as seen by the abstract machine.)

- $G_{i,A}$ describes the guarantee condition from component i towards the environment.
  Note: this is the same as what the environment should rely on for component i to provide, which we denote by $R_{i,A}$.

- $R_{A,i}$ describes the rely condition of component i w.r.t. the environment.
  Note: this is the same as what the environment should guarantee towards component i, which we denote by $G_{A,i}$.

- $G_{i,j}$ describes the internal relation, i.e. what component i guarantees to component j.
  Note: this is the same as what component j should rely on component i to provide, which we denote by $R_{i,j}$.

If we use this split, the proof obligations of the rule change to:

- $\mathcal{P} \models \bigwedge_i G_{i,A} \to G_A, \mathcal{P} \models G_i \to [G_{i,A} \wedge \bigwedge_{j \neq i} G_{i,j}]$
  (this was $\mathcal{P} \models \bigwedge_i G_i \to G_A$)

- $\mathcal{P} \models R_A \to \bigwedge_i R_{A,i}, \mathcal{P} \models [R_{A,i} \wedge \bigwedge_{j \neq i} R_{j,i}] \to R_i$
  (this was $\mathcal{P} \models [R \wedge \bigwedge_{j \neq i} G_j] \to R_i$)

The resulting rule has still the same trouble as the previous one, though in a less trivial form. This is illustrated in the next example.

Example

Suppose there are two component machines. Suppose we have proven the following proof obligations:

1. $\mathcal{P} \models R_1 \to G_1$
2. $\mathcal{P} \models R_2 \to G_2$
3. $\mathcal{P} \models R_A \to [R_{A,1} \wedge R_{A,2}]$
4. $\mathcal{P} \models G_1 \to [G_{1,A} \wedge G_{1,2}]$
5. $\mathcal{P} \models G_2 \to [G_{2,A} \wedge G_{2,1}]$
6. $\mathcal{P} \models [G_{1,A} \wedge G_{2,A}] \to G_A$
7. $\mathcal{P} \models [R_{A,1} \wedge R_{2,1}] \to R_1$
8. $\mathcal{P} \models [R_{A,2} \wedge R_{1,2}] \to R_2$

We want to infer $\mathcal{P} \models R_A \to G_A$ from 1-8.
Assume R_A holds then we must prove that G_A holds.
From 6 we can conclude: $G_{1,A} \wedge G_{2,A}$ must hold.
From 4 and 5 we conclude: $G_1 \wedge G_2$ must hold.
From 1 and 2 we conclude: $R_1 \wedge R_2$ must hold.
From 7 and 8 we conclude: $[R_{A,1} \wedge R_{2,1}] \wedge [R_{A,2} \wedge R_{1,2}]$ must hold.

From 3 we conclude: $R_A \wedge R_{2,1} \wedge R_{1,2}$ must hold.
From assumption we conclude: $R_{2,1} \wedge R_{1,2}$ must hold.
Because of $R_{2,1} \equiv G_{2,1}$ and $R_{1,2} \equiv G_{1,2}$: $G_{2,1} \wedge G_{1,2}$ must hold.
From 4 and 5 we conclude: $G_1 \wedge G_2$ must hold.
So we get cycle $G_1 \wedge G_2 \rightarrow R_1 \wedge R_2 \rightarrow R_{2,1} \wedge R_{1,2} \rightarrow G_1 \wedge G_2$.

end example

Now unsoundness can be shown similarly as before. The proof obligations in the cycle can again be trivially fulfilled by choosing false for all assertions involved. It is than also possible to choose R_A and all $R_{A,i}$ to be true and G_A to be false. $\mathcal{P} \models R_A \rightarrow G_A$ is than again derivable, and the rule therefore unsound.

Stark's solution to the cycle problem is given for this version of the rule. The idea is to require that the set of $G_{i,j}$ is acyclic, i.e. to require that there can be no unbroken cyclic dependency between components.
Formally: $\{G_{i,j} : i, j \in I\}$ is acyclic if
$\mathcal{P} \models \bigvee_{k=0}^{n-1} G_{i_k,i_{k+1}}$ for all simple cycles $\{(i_0, i_1), \ldots, (i_{n-1}, i_n)\}$ in I with $i_n = i_0$.
This additional information breaks the circular reasoning. This means for the above example that we have the extra proof obligation $\mathcal{P} \models G_{1,2} \vee G_{2,1}$. The last example used above illustrates how this extra proof obligation works.

Example

Suppose there are two component machines and that we have proven the proof obligations as before, together with:

9. $\mathcal{P} \models G_{1,2} \vee G_{2,1}$

Suppose R_A holds.
From 3 we infer: $R_{A,1} \wedge R_{A,2}$ holds.
From 7 and 8 and logical reasoning we infer: $(R_{2,1} \rightarrow R_1) \wedge (R_{1,2} \rightarrow R_2)$.
From 1 and 2 we infer: $(R_{2,1} \rightarrow G_1) \wedge (R_{1,2} \rightarrow G_2)$.
From 4 and 5 we infer: $(R_{2,1} \rightarrow [G_{1,A} \wedge G_{1,2}]) \wedge (R_{1,2} \rightarrow [G_{2,A} \wedge G_{2,1}])$.
With logical reasoning we infer:
$(R_{2,1} \rightarrow G_{1,2}) \wedge (R_{1,2} \rightarrow G_{2,1}) \wedge (R_{2,1} \rightarrow G_{1,A}) \wedge (R_{1,2} \rightarrow G_{2,A})$. (*)
The first two form the cycle we are discussing because
$G_{1,2} \equiv R_{1,2}$ and $G_{2,1} \equiv R_{2,1}$: $R_{2,1} \rightarrow G_{1,2} \equiv R_{1,2} \rightarrow G_{2,1} \equiv R_{2,1}$.
This cycle is broken by condition 9.
From 9 and $R_{2,1} \rightarrow G_{1,2}$ we infer with logical reasoning: $R_{1,2}$.
From 9 and $R_{1,2} \rightarrow G_{2,1}$ we infer with logical reasoning: $R_{2,1}$.
So (*) becomes: $R_{2,1} \wedge R_{1,2} \wedge (R_{2,1} \rightarrow G_{1,A}) \wedge (R_{1,2} \rightarrow G_{2,A})$.
With logical reasoning we infer: $G_{1,A} \wedge G_{2,A}$.
From 6 we infer: G_A.
So from 1-9 we can infer: $R_A \rightarrow G_A$.

end example

In [12, 13] Stark gives a soundness proof of this last rule.

2.8 Relationship with Lamport's model

The relationship between Lamport's model and the one of Stark is mainly the way how the stutter-problem is solved. This problem is as follows. Given two observations of a system, the first observation contains only consecutive snap-shots of the system that differ from each other whereas the second observation contains the same snap-shots but also some consecutive ones that are identical. This is called stuttering. Clearly these observations must be considered to be equal. The problem is: how must we do that? Both methods provide a solution. In [3] these solutions are discussed in detail, here we only give an informal description of them.

Lamport's methods [7, 6, 1] use infinite discrete state sequences as model of observations. Because of this discrete time domain, a temporal operator referring to the next state can be (and in many temporal logics is) defined. Specifications should not distinguish between sequences that are equal modulo stuttering. Therefore the use of this operator in specifications is simply forbidden.

Stark's method [12, 13, 14] uses dense time models, in which an execution is modeled by a state-valued function of the set of non-negative reals. Using dense time is based on the intuition that state changes happen only now and then, so that in between two consecutive changes there are uncountable moments at which *nothing* happens. Consequently, it is impossible to count, or express, stutter-steps, i.e., there is no next state operator.

In both frameworks there is a completeness problem if refinement mappings or relations are used to prove correctness. Intuitively, this is connected with the amount of information present in states. Abadi and Lamport, in [1], present a solution for the discrete framework, using history as well as prophesy variables, that can be also used in the dense setting.

2.9 Our use of Stark's formalism

In Stark's formalism a separation is made between the machine part (local) and the validity condition part (global), see Section 2.5. We use the validity condition part not for liveness but for deleting undesirable sequences. In the readers/writers example we will see that a deadlocked sequence is an example of such an undesirable sequence. These deadlocked sequences are removed by defining the proper validity condition. In the next section we see what such a validity condition looks like.

3 R/W-Problem in Stark's Formalism

We are now ready to apply Stark's formalism to Dijkstra's development. The readers/writers problem, described intuitively, is as follows: given N readers and M writers, a reader performs, cyclically, non-critical action `NCS` and critical action `READ`, and a writer performs, again cyclically, non-critical action `NCS` and critical action `WRITE`. We must synchronise these readers and writers in such a way that if a writer performs the `WRITE` action it is the only process that performs a critical action, i.e. mutual exclusion is required (ME). Furthermore, it is necessary that any request to execute the critical action is eventually granted, i.e. eventual access should hold (EA). It is this synchroniser that has to be developed. But before we give the development we formulate an abstract specification for the problem.

The development process has four steps: in the first step Dijkstra gives an implementation by a program that produces undesirable deadlocked sequences. In the second step Dijkstra uses the split binary semaphore technique to delete the deadlocked sequences from the first implementation; he obtains by this technique a second implementation that introduces as undesirable sequences new deadlocked ones. These deadlocked sequences are deleted in the third step resulting in a third implementation that contains as undesirable sequences unnecessarily blocking ones. These sequences are not deadlocking sequences but only sequences that are inefficient because they suspend a reader or writer unnecessarily. In the fourth step, these sequences are deleted and also the resulting implementation is cleaned up.

3.1 The abstract specification

We follow [2] and show how the informal approach used there can be formalised. Dijkstra rewrites the informal specification as follows: as a first step, he describes readers and writers by programs (whose semantics he assumes are intuitively clear):

$$\texttt{reader0:}\quad \textbf{do}\ \texttt{true} \rightarrow \texttt{NCS;READ}\ \ \textbf{od}$$

$$\texttt{writer0:}\quad \textbf{do}\ \texttt{true} \rightarrow \texttt{NCS;WRITE}\ \textbf{od}$$

He then combines these programs into one parallel program `S0`. `S0` forms the abstract specification and is defined as follows:

$$\texttt{S0 :}\quad \|_{i=1}^{N}\ \texttt{reader0}\ \|\ \|_{j=1}^{M}\ \texttt{writer0}\ ,$$

Where $\|_{i=1}^{N}$`reader0` is a notation for the N-fold parallel composition of `reader0`.

Finally he formulates an informal requirement to exclude from `S0` the unwanted sequences. This requirement is the same as in the introduction: ME and EA. The complete abstract specification is thus `S0` plus this requirement.

We transform `S0` into a machine M_A and the informal requirements ME and EA into V_A to get a specification a la Stark. Note: `S0` has some liveness or fairness property that is assumed a priori by Dijkstra. Which property Dijkstra assumes is not entirely clear from [2]. We assume, following Stark, that all machines have the property that if a machine is infinitely often enabled it will infinitely often make a move. This corresponds to strong fairness.

We want to specify the behaviour of the synchroniser module in an environment of readers and writers. As it is really the scheduler we wish to specify, it seems advantageous to us to single this part out as a separate component. The specification $S_A = \langle M_A, V_A \rangle$ we shall use is then as follows.

1. **Events:**
$E_A = \{rtryi{\downarrow}, rruni{\uparrow}, rresti{\downarrow}, wtryj{\downarrow}, wrunj{\uparrow}, wrestj{\downarrow}:$
$i \in [1, \ldots, N], j \in [1, \ldots, M]\}$
When a *rtryi* event occurs M_A knows that *readeri* wants to execute `READ`. M_A subsequently generates a *rruni* event to signal *readeri* that it may execute `READ`. When a *rresti* event occurs M_A knows that *readeri* has finished executing `READ`. Similarly for *writerj*.

2. **States:**
$Q_A : (\{r1, \ldots, rN\} \cup \{w1, \ldots, wM\}) \rightarrow \{tryg, rung, resg, err\}$
$\mathbf{st}(ri) = tryg$: *readeri* wants to execute `READ`.
$\mathbf{st}(ri) = err$: *readeri* is not functioning correctly. Note, that this is the state of the scheduler, reflecting the activities of the readers and writers.

3. **Initial States:**
$IQ_A \equiv \bigwedge_{i=1}^{N} \mathbf{st}(ri) = resg \wedge \bigwedge_{j=1}^{M} \mathbf{st}(wj) = resg$

4. **Transitions:**
$TR_A \equiv$

   - $\mathbf{e} = rtryi{\downarrow} \wedge((\mathbf{st}(ri) = resg \wedge \mathbf{st'} = \mathbf{st} \mid ri : tryg) \vee$
   $(\mathbf{st}(ri) \neq resg \wedge \mathbf{st'} = \mathbf{st} \mid ri : err))$
   If *readeri* is functioning correctly (i.e. the synchroniser is not in the *err* state for this reader) then it goes from state *resg* to state *tryg* on the occurrence of the *rtryi* event. This event signals the synchroniser that *readeri* wants to to execute its `READ`.
   - $\mathbf{e} = rruni{\uparrow} \wedge \mathbf{st}(ri) = tryg \wedge \mathbf{st'} = \mathbf{st} \mid ri : rung$
   When a reader has signaled the synchroniser that it wants to execute `READ`, the synchroniser signals with a *rruni* event that it may execute its `READ`. Note: because we follow Dijkstra the synchroniser does not check if there are writers that are currently execute their `WRITE`. We have could have done it here but then ME can be dropped from the validity conditions because it is then already specified here.
   - $\mathbf{e} = rresti{\downarrow} \wedge((\mathbf{st}(ri) = rung \wedge \mathbf{st'} = \mathbf{st} \mid ri : resg) \vee$
   $(\mathbf{st}(ri) \neq rung \wedge \mathbf{st'} = \mathbf{st} \mid ri : err))$
   When *readeri* has finished executing `READ`, it signals this to the synchroniser with a *rresti* event.

   The writer events can be dealt with in the same way.

5. **Validity Conditions:**
V_A extracts from $Comp(M_A)$ those sequences that satisfy the mutual exclusion requirement ME: when a writer executes its `WRITE` then no other writers are executing `WRITE` and no readers executing `READ`. And V_A also extracts those sequences that satisfy the "liveness" requirement EA: when a reader or writer wants to execute its critical section it is

eventually allowed to do so. Formally:

$V_A \equiv P_0 \wedge R_A \rightarrow G_A$

$P_0 \equiv \Box((\bigwedge_{j=1}^{M} \mathbf{st}(wj) \neq rung)\vee$
$((\sum_{j=1}^{M} \mathbf{st}(wj) = rung) = 1 \wedge \bigwedge_{i=1}^{N} \mathbf{st}(ri) \neq rung))$

This is the ME requirement.

$R_A \equiv \Box(\bigwedge_{i=1}^{N}(\mathbf{st}(ri) = rung \rightarrow \Diamond(\mathbf{st}(ri) = resg))\wedge$
$\bigwedge_{j=1}^{M}(\mathbf{st}(wj) = rung \rightarrow \Diamond(\mathbf{st}(wj) = resg)))$

$G_A \equiv \Box(\bigwedge_{i=1}^{N}(\mathbf{st}(ri) = tryg \rightarrow \Diamond(\mathbf{st}(ri) = rung))\wedge$
$\bigwedge_{j=1}^{M}(\mathbf{st}(wj) = tryg \rightarrow \Diamond(\mathbf{st}(wj) = rung)))$

And this the EA requirement.

3.2 The first development step

Dijkstra's next step is to translate the informally stated requirement into formal program form, i.e. to transform `reader0` and `writer0` in such a way that they satisfy the synchronisation requirement ME. We discuss this translation informally.

He introduces shared variables `aw` and `ar` and binary semaphore `mx`. Shared variable `ar` represents the number of readers which may execute their `READ`, and `aw` represents the number of writers which may execute their `WRITE`. A reader increases `ar` by 1 if it is allowed to execute its `READ` and decreases `ar` by 1 if it is finished with executing its `READ`. Since `ar` will be changed and accessed by several readers, Dijkstra protects the operation of increasing and decreasing `ar` by semaphore operations P and V on binary semaphore mx to ensure that only one reader changes `ar` at a time, i.e. mutual exclusion. The synchronisation requirement is brought into `reader0` by guarding the increasing operation of `ar` with condition `aw=0`, i.e., the number of writers that may execute their `WRITE` equals zero. The same can be done for `writer0`. The initial values of the shared variables are 0 and the initial value of semaphore `mx` is 1. This results in the following programs:

```
reader1:
     do true → NCS;
               P(mx);(*) if aw=0 →ar:=ar+1 fi;V(mx);
               READ;
               P(mx);ar:=ar-1;V(mx)
     od
writer1:
     do true → NCS;
               P(mx);(+) if aw=0 ∧ ar=0→aw:=aw+1 fi;V(mx);
               WRITE;
               P(mx);aw:=aw-1;V(mx)
     od
```

S1 : $\|_{i=1}^{N}$ reader1 $\|$ $\|_{j=1}^{M}$ writer1

Dijkstra now formulates a requirement for this collection of programs. This is necessary because this collection can generate new unwanted sequences, namely sequences which can deadlock. One such sequence is for instance:

> A writer starts in the initial state and then executes `NCS;P(mx);(+)`, as result of this the value of `aw` changes in 1. A reader then executes `NCS;P(mx);(*)` and blocks in the `if-fi` clause of `(*)` because `aw=1` and the semantics of this `if-fi` is such that when no guard is fulfilled it blocks. Then no reader or writer can then execute `(*)` or `(+)` because `mx=0` and `mx` holds this value forever. The requirement is thus that these deadlocked sequences are not generated.

(Note: `S0` generates no deadlocked sequences, so `S1` generates some sequences that `S0` did not generate. The deadlocked sequences that are generated by the machine corresponding to `S1` are removed by the validity condition corresponding to `S1`, so that the set of allowed sequences of `S1` is not bigger than that of `S0`.)

We again specify `S1` plus the requirement that no deadlocked sequences are allowed in Stark's formalism. This specification must implement S_A. In Stark's formalism an implementation consists of the interconnection, the abstract specification and the component specifications. The abstract specification is S_A. We have seen that `S1` uses variables `ar,aw` and semaphore `mx`. These variables correspond to components ar, aw and mx in Stark's formalism. The PV-segments of `reader1` and `writer1` correspond to components $rn1$ and $wn1$. These are the components that take care of the synchronisation. In the next subsections we show how these component specifications are formulated in Stark's formalism.

3.2.1 Specification of a shared variable

We give a specification of a general shared variable with initial value K. Informally the specification is that the environment retrieves the current value of the shared variable with a $g(v)$ event and updates it with a $p(w)$ event. The formal specification $Ssv_K = \langle Msv_K, Vsv_K \rangle$ is as follows:

1. **Events:**
   $Esv_K = \{g(v)\!\downarrow, p(w)\!\downarrow : v, w \in Z\}$

2. **States:**
   $Qsv_K : svs \rightarrow Z$
   $\mathbf{st}(svs) = z$: the current value of the shared variable is z.

3. **Initial States:**
   $IQsv_K \equiv \mathbf{st}(svs) = K$

4. **Transitions:**
   $TRsv_K \equiv$

- $\mathbf{e} = g(v){\downarrow} \wedge v = \mathbf{st}(svs) \wedge \mathbf{st}' = \mathbf{st}$
  The environment retrieves the current value of the shared variable.
- $\mathbf{e} = p(w){\downarrow} \wedge \mathbf{st}' = \mathbf{st} \mid svs : w$
  The environment updates the current value of the shared variable.

5. **Validity Conditions:**
   All the sequences Msv_K generates are allowed, so: $Vsv_K \equiv true$.
   (In R/G form this is true→ true, i.e. no liveness requirements are imposed.)

3.2.2 The specification of a binary semaphore

We give an abstract specification of a general binary semaphore with initial value K. Informally this specification is as follows. A component that uses this semaphore signals with a tPi event that it wants to execute its P-operation. The semaphore signals with an ePi event that this component may execute its P-operation. The component signals with a Vi event that it has executed the V-operation. Note, that the validity conditions formalise the as-yet-unformalised concept of fairness used in the programs in [2]. (As will become clear later, a strong semaphore is used.)
The formal specification $SsemA_K = \langle MsemA_K, VsemA_K \rangle$ is as follows:

1. **Events:**
   $EsemA_K = \{tPi{\downarrow}, ePi{\uparrow}, Vi{\downarrow} \colon i \in \{1, \ldots, H\}\}$
   (H is the number of components using the binary semaphore.)

2. **States:**
   $QsemA_K : sems : \{sem0, sem1, err\} \times wset : \{1, \ldots, H\}$
   $\mathbf{st}(sems) = sem0$:
   A P-operation corresponds with a decrease of 1 and a V-operation corresponds with an increase of 1, so $semi$ corresponds with value i. The variable $wset$ denotes the set of indices of the components that are waiting to execute P.

3. **Initial States:**
   $IQsemA_K \equiv \mathbf{st}(sems) = semK \wedge \mathbf{st}(wset) = \emptyset$

4. **Transitions:**
   $TRsemA_K \equiv$

   - $\mathbf{e} = tPi{\downarrow} \wedge ((i \notin \mathbf{st}(wset) \wedge \mathbf{st}' = \mathbf{st} \mid wset : \mathbf{st}(wset) \bigcup \{i\}) \vee$
     $(i \in \mathbf{st}(wset) \wedge \mathbf{st}' = \mathbf{st} \mid sems : err))$
     Component i wants to execute a P-operation on the semaphore. If component i is not in the waiting set it will be inserted.
   - $\mathbf{e} = ePi{\uparrow} \wedge \mathbf{st}(sems) = sem1 \wedge i \in \mathbf{st}(wset) \wedge$
     $\mathbf{st}' = \mathbf{st} \mid sems, wset : sem0, \mathbf{st}(wset)/\{i\}$
     The semaphore only generates an ePi event if the value of the semaphore equals one and component i has generated a tPi event before.

- $\mathbf{e} = Vi{\downarrow} \wedge ((\mathbf{st}(sems) = sem0 \wedge \mathbf{st}' = \mathbf{st} \mid sems : sem1) \vee$
  $(\mathbf{st}(sems) \neq sem0 \wedge \mathbf{st}' = \mathbf{st} \mid sems : err))$
  Component i generates a V-operation on the semaphore.

5. **Validity Conditions:**
   With $VsemA_K$ we can express the liveness properties of a semaphore. $VsemA_K$ must specify which sequences, that $MsemA_K$ generates, are allowed. This is needed because $MsemA_K$ can generate sequences in which a component i never finishes its P-operation. We express with $VsemA_K$ that $MsemA_K$ is a strong semaphore because Dijkstra apparently also uses a strong semaphore in his implementation.
   $VsemA_K \equiv RsemA_K \rightarrow GsemA_K$
   $RsemA_K \equiv \Box(\bigwedge_{i=1}^{H}(\mathbf{e} = ePi \rightarrow \Diamond(\mathbf{e} = Vi)))$
   $GsemA_K \equiv \Box(\bigwedge_{i=1}^{H}(\mathbf{e} = tPi \rightarrow \Diamond(\mathbf{e} = ePi)))$
   $MsemA_K$ relies on the environment to generate a Vi event if it has generated an ePi event itself. $MsemA_K$ guarantees then if the environment generates a tPi event that it eventually generates an ePi event.

3.2.3 Specification of component $rn1$

We now give the specification of component $rn1$ (the specification of $wn1$ is analogous). Component $rn1$ corresponds to the PV-segments of `reader1`.
The specification $Srn1 = \langle Mrn1, Vrn1 \rangle$ is as follows:

1. **Events:**
   $Ern1 = EV \bigcup \{Vmx{\uparrow}, ePmx{\downarrow}, tPmx{\uparrow}\}$
   where $EV = \{try{\downarrow}, run{\uparrow}, rest{\downarrow}, gaw(w){\uparrow}, gar(v){\uparrow}, par(u){\uparrow} : u, v, w \in N\}$

2. **States:**
   $Qrn1 : rs : RS1 \times rr : N \times rw : N$
   where $RS1 = \{resg, tryg, tPV1, iPV1, gaw1, gar1, par1, aPV1, rung,$
   $bPV2, tPV2, iPV2, gar2, par2, err, err1\}$

3. **Initial States:**
   $IQrn1 \equiv \mathbf{st}(rs) = resg$

4. **Transitions:**
   $TRrn1 \equiv$

   1 $\mathbf{e} = try{\downarrow} \wedge ((\mathbf{st}(rs) = resg \wedge \mathbf{st}' = \mathbf{st} \mid rs : tryg) \vee$
   $(\mathbf{st}(rs) \neq resg \wedge \mathbf{st}' = \mathbf{st} \mid rs : err))$
   The reader signals with a $rtry$ event to $rn1$ that it wants to execute `READ`.

   2 $\mathbf{e} = tPmx{\uparrow} \wedge ((\mathbf{st}(rs) = tryg \wedge \mathbf{st}' = \mathbf{st} \mid rs : tPV1) \vee$
   $(\mathbf{st}(rs) = bPV2 \wedge \mathbf{st}' = \mathbf{st} \mid rs : tPV2))$
   $rn1$ requests with a $tPmx$ event that it wants to enter a PV-section, that is, it wants access to the components ar and aw.

3 $\mathbf{e} = ePmx\!\uparrow \wedge$
$((\mathbf{st}(rs) = tPV1 \wedge \mathbf{st}' = \mathbf{st} \mid rs : iPV1) \vee$
$(\mathbf{st}(rs) = tPV2 \wedge \mathbf{st}' = \mathbf{st} \mid rs : iPV2) \vee$
$((\mathbf{st}(rs) \neq tPV1 \vee \mathbf{st}(rs) \neq tPV2) \wedge \mathbf{st}' = \mathbf{st} \mid rs : err1))$
The mx component signals with a $ePmx$ event that $rn1$ may enter its PV-section and thus has access to components ar and aw.

4 $\mathbf{e} = gaw(w)\!\uparrow \wedge \mathbf{st}(rs) = iPV1 \wedge \mathbf{st}' = \mathbf{st} \mid rs, rw : gaw1, w$
The synchroniser retrieves the current value of aw.

5 $\mathbf{e} = gar(v)\!\uparrow \wedge((\mathbf{st}(rs) = gaw1 \wedge \mathbf{st}' = \mathbf{st} \mid rs, rr : gar1, v) \vee$
$(\mathbf{st}(rs) = iPV2 \wedge \mathbf{st}' = \mathbf{st} \mid rs, rr : gar2, v))$
The synchroniser retrieves the current value of ar.

6 $\mathbf{e} = par(u)\!\uparrow \wedge$
$((u = \mathbf{st}(rr) + 1 \wedge \mathbf{st}(rs) = gar1 \wedge \mathbf{st}(rw) = 0 \wedge$
$\mathbf{st}' = \mathbf{st} \mid rs, rr : par1, u) \vee$
$(u = \mathbf{st}(rr) - 1 \wedge \mathbf{st}(rs) = gar2 \wedge \mathbf{st}' = \mathbf{st} \mid rs, rr : par2, u))$
If $rn1$ is in the first PV-section then it increases component ar with one if the current value of the aw component is zero. That means that there are no writers executing their `WRITE`. If the current value of aw is not zero, component $rn1$ will be deadlocked in its PV-section.
If $rn1$ is in the second PV-section then it decreases ar with one.

7 $\mathbf{e} = Vmx\!\uparrow \wedge((\mathbf{st}(rs) = par1 \wedge \mathbf{st}' = \mathbf{st} \mid rs : aPV1) \vee$
$(\mathbf{st}(rs) = par2 \wedge \mathbf{st}' = \mathbf{st} \mid rs : resg))$
After updating the ar component $rn1$ signals with a Vmx event that it leaves its PV-section.

8 $\mathbf{e} = run\!\uparrow \wedge \mathbf{st}(rs) = aPV1 \wedge \mathbf{st}' = \mathbf{st} \mid rs : rung$
When $rn1$ has passed first PV-section it signals with an $rrun$ event to its corresponding reader that it may execute `READ`.

9 $\mathbf{e} = rest\!\downarrow \wedge((\mathbf{st}(rs) = rung \wedge \mathbf{st}' = \mathbf{st} \mid rs : bPV2) \vee$
$(\mathbf{st}(rs) \neq rung \wedge \mathbf{st}' = \mathbf{st} \mid rs : err))$
The reader signals with a $rest$ event $rn1$ that it has finished `READ`.

5. **Validity Conditions:**
The set $Vrn1$ of allowed sequences of $Mrn1$ is as follows:
$Vrn1 \equiv Rrn1 \rightarrow Grn1$
$Rrn1 \equiv$

- $\Box(\mathbf{st}(rs) = rung \rightarrow \Diamond(\mathbf{st}(rs) = bPV2))$
$rn1$ relies on its reader that the execution of `READ` takes only a finite amount of time.
- $\wedge\Box(\mathbf{e} = tPmx \rightarrow \Diamond(\mathbf{e} = ePmx))$
$rn1$ furthermore relies on mx that it eventually gives the access-right if $rn1$ has asked for it.

$Grn1 \equiv$

- $\Box(\mathbf{st}(rs) = tryg \rightarrow \Diamond(\mathbf{st}(rs) = rung))$
  $rn1$ then guarantees to its reader that it eventually may execute `READ` if the reader has asked for it.
- $\wedge\Box(\mathbf{e} = ePmx \rightarrow \Diamond(\mathbf{e} = Vmx))$
  $rn1$ furthermore guarantees to mx that it has the access-right only a finite amount of time.

These last two conditions remove the unwanted deadlocked sequences because it is required that when the $rn1$ component gets in its PV-segment it must eventually leave this PV-segment, i.e., not get deadlocked in it.

3.2.4 Correctness of the implementation

The implementation is correct if the maximality and the validity conditions hold. This means that we have to prove the following:
(1) Any event that can be generated by the system of component machines can also be performed by the abstract machine.
maximality:

$$Comp(M_c) \models \forall e \in E_c : (Reachable_c \wedge \bigwedge_{i=1}^{N}[Enabled_{rn1}(e)]_{ri\mathbf{toc}} \wedge \bigwedge_{j=1}^{M}[Enabled_{wn1}(e)]_{wj\mathbf{toc}} \wedge [Enabled_{sv0}(e)]_{ar\mathbf{toc}} \wedge [Enabled_{sv0}(e)]_{aw\mathbf{toc}} \wedge [Enabled_{semA1}(e)]_{mx\mathbf{toc}}) \rightarrow [Enabled_A(e)]_{A\mathbf{toc}},$$

where E_c denotes the interface of the composite machine. The proof that this formula holds is not difficult but rather long so we present do not present it.
(2) Any allowed computation of each component machine corresponds with an allowed computation of the abstract machine.
validity:

$$Comp(M_c) \models (\bigwedge_{i=1}^{N}[Vrn1]_{rn1\mathbf{toc}} \wedge \bigwedge_{j=1}^{M}[Vwn1]_{wn1\mathbf{toc}} \wedge [VsemA_1]_{mx\mathbf{toc}} \wedge [Vsv_0]_{aw\mathbf{toc}} \wedge [Vsv_0]_{ar\mathbf{toc}}) \rightarrow [V_A]_{A\mathbf{toc}},$$

where M_c denotes the composite machine. The proof of this can be done with the rely/guarantee rule and is not difficult but it is again too long to present it here.

3.3 The second development step

The components of the first implementation still generate sequences, i.e. deadlocked ones, which are not allowed by the validity conditions of these components. In this step we change components $rn1$ and $wn1$ because these components are responsible for the generation of these deadlocked sequences. This is the same as is done by Dijkstra: he massages `reader1` and `writer1` into `reader2` and `writer2` so that no deadlocked sequences inside a PV-segment are generated any more.

One such deadlocked sequence generated by the first implementation is as follows: suppose $rn1$ has gained the access-right for the shared variables (first

PV-segment) and has executed $gar(v)$ and $gaw(w)$; suppose also $w = 1$ (a writer is executing `WRITE`). Then $rn1$ can never execute the $par(\mathbf{st}(rar) + 1)$ event , i.e., $rn1$ has deadlocked. This sequence is not allowed by $Vrn1$ because $rn1$ must guarantee that if it gets the access-right it must eventually give it back.

Dijkstra uses the split binary semaphore technique to prevent programs from becoming deadlocked inside a PV-segment. The idea is that we must prevent programs from getting the access-right (get into a PV-segment) for the shared variables if we know that they can not give it back (get deadlocked inside a PV-segment). For `reader1` this means: never let it enter the first PV-segment if `aw` does not equal zero. For `writer1` this means: never let it enter the first PV-segment if `aw` or `ar` does not equal zero. `reader1` and `writer1` never block in their second PV-segment.
How does one prevent that `reader1` gets deadlocked inside a PV-segment? This is done as follows: `reader1` chooses, when it gives the access-right back, who can have it thereafter. `Reader1` executes therefore the following piece of program as replacement for `V(mx)`:

```
CHOOSE: if true → V(m) [] aw=0 → V(r) [] aw=0 ∧ ar=0 → V(w) fi
```

We have to split `mx` in three pieces. If `aw` equals zero then a reader is allowed to enter its first PV-segment, i.e., this PV-segment is not guarded by `P(mx)` but by `P(r)`. We do this substitution for all PV-segments of `reader1` and `writer1`. So we have replaced `mx` by three other binary semaphores.
What is the initial value of these semaphores? If they all have initial value 1 then more than one program can have access-right to the shared variables, i.e., only one has initial value 1. Semaphore `r` can not have initial value 1 because if no reader wants to execute `READ` then no writer can execute `WRITE`. The same holds for semaphore `w`. Thus `m` has initial value 1. But then no reader or writer can enter the first PV-segment. The solution to this problem is that we insert a PV-segment (`P(m);CHOOSE`) in front of the first one. This is in short what Dijkstra does to prevent that `reader1` and `writer1` get deadlocked inside a PV-segment. The result of this transformation is:

```
reader2:
            do true → NCS;
                      P(m);CHOOSE;
                      P(r);ar:=ar+1;CHOOSE;
                      READ;
                      P(m);ar:=ar-1;CHOOSE
            od
writer2:
            do true → NCS;
                      P(m);CHOOSE;
                      P(w);aw:=aw+1;CHOOSE;
                      WRITE;
```

```
                    P(m);aw:=aw-1;CHOOSE
          od
S2 :  ||_{i=1}^{N} reader2 || ||_{j=1}^{M} writer2
```

`S2` generates no sequences that can deadlock inside a PV-segment. But `S2` can generate sequences that can deadlock outside these segments, e.g. initially `reader2` can choose for a `V(w)` operation, and get blocked by a `P(r)` operation. Then no other reader or writer can enter the first PV-segment because `m` equals zero. The informal requirement is thus that no such sequences are allowed.

3.3.1 Specification of component rn2

The result of the split binary semaphore technique is that $rn1$ (and $wn1$) have to be changed because they are accessing now three semaphores instead of one. So semaphore mx has to replaced by semaphores m, r and w. We give the changes of component $rn1$, i.e., $Srn1$ changes to $Srn2 = \langle Mrn2, Vrn2\rangle$. (Note: we have numbered the transitions in the first implementation. These numbers correspond with the numbers in the following implementation, for instance, transition 2 of the first implementation is replaced by transitions 2.1 and 2.2 in the second implementation.)

1. **Events:**
   $Ern2 = EV \bigcup \{Vm\uparrow, ePm\downarrow, tPm\uparrow, Vr\uparrow, ePr\downarrow, tPr\uparrow, Vw\uparrow\}$
2. **States:**
   $Qrn2 : rs : RS2 \times rr : N \times rw : N$
   where $RS2 = RS1 \bigcup \{tPV0, iPV0, gaw0, gar0, aPV0, gaw2\}$
3. **Initial States:**
   $IQrn2 \equiv \mathbf{st}(rs) = resg$
4. **Transitions:**
   $TRrn2 \equiv$

   1 same as $TRrn1$

   2.1 $\mathbf{e} = tPm\uparrow \wedge((\mathbf{st}(rs) = tryg \wedge \mathbf{st'} = \mathbf{st} \mid rs : tPV0)\vee$
   $(\mathbf{st}(rs) = bPV2 \wedge \mathbf{st'} = \mathbf{st} \mid rs : tPV2))$
   The synchroniser signals with a tPm event that it wants to enter the first or third PV-segment.

   2.2 $\mathbf{e} = tPr\uparrow \wedge \mathbf{st}(rs) = aPV0 \wedge \mathbf{st'} = \mathbf{st} \mid rs : tPV1$
   The synchroniser signals with a tPr event that it wants to enter the second PV-segment.

   3.1 $\mathbf{e} = ePm\downarrow \wedge$
   $((\mathbf{st}(rs) = tPV0 \wedge \mathbf{st'} = \mathbf{st} \mid rs : iPV0)\vee$
   $(\mathbf{st}(rs) = tPV2 \wedge \mathbf{st'} = \mathbf{st} \mid rs : iPV2)\vee$
   $((\mathbf{st}(rs) \neq tPV0 \vee \mathbf{st}(rs) \neq tPV2) \wedge \mathbf{st'} = \mathbf{st} \mid rs : err1))$
   The synchroniser may enter the first or third PV-segment.

3.2 $\mathbf{e} = ePr{\downarrow} \wedge((\mathbf{st}(rs) = tPV1 \wedge \mathbf{st}' = \mathbf{st} \mid rs : iPV1) \vee$
$(\mathbf{st}(rs) \neq tPV1 \wedge \mathbf{st}' = \mathbf{st} \mid rs : err1))$
The synchroniser may enter the second PV-segment.

4 $\mathbf{e} = gaw(w){\uparrow} \wedge((\mathbf{st}(rs) = iPV0 \wedge \mathbf{st}' = \mathbf{st} \mid rs, rw : gaw0, w) \vee$
$(\mathbf{st}(rs) = iPV1 \wedge \mathbf{st}' = \mathbf{st} \mid rs, rw : gaw1, w) \vee$
$(\mathbf{st}(rs) = iPV2 \wedge \mathbf{st}' = \mathbf{st} \mid rs, rw : gaw2, w))$
The synchroniser retrieves the current value of aw.

5 $\mathbf{e} = gar(v){\uparrow} \wedge((\mathbf{st}(rs) = gaw0 \wedge \mathbf{st}' = \mathbf{st} \mid rs, rr : gar0, v) \vee$
$(\mathbf{st}(rs) = gaw1 \wedge \mathbf{st}' = \mathbf{st} \mid rs, rr : gar1, v) \vee$
$(\mathbf{st}(rs) = gaw2 \wedge \mathbf{st}' = \mathbf{st} \mid rs, rr : gar2, v))$
The synchroniser retrieves the current value of ar.

6 $\mathbf{e} = par(u){\uparrow} \wedge$
$((u = \mathbf{st}(rar) + 1 \wedge \mathbf{st}(rs) = gar1 \wedge \mathbf{st}' = \mathbf{st} \mid rs, rr : par1, u) \vee$
$(u = \mathbf{st}(rar) - 1 \wedge \mathbf{st}(rs) = gar2 \wedge \mathbf{st}' = \mathbf{st} \mid rs, rr : par2, u))$
The synchroniser increases or decreases ar.

7.1 $\mathbf{e} = Vm{\uparrow} \wedge((\mathbf{st}(rs) = gar0 \wedge \mathbf{st}' = \mathbf{st} \mid rs : aPV0) \vee$
$(\mathbf{st}(rs) = par1 \wedge \mathbf{st}' = \mathbf{st} \mid rs : aPV1)) \vee$
$(\mathbf{st}(rs) = par2 \wedge \mathbf{st}' = \mathbf{st} \mid rs : resg))$
The synchroniser chooses the Vm branch.

7.2 $\mathbf{e} = Vr{\uparrow} \wedge((\mathbf{st}(rs) = gar0 \wedge \mathbf{st}(rw) = 0 \wedge \mathbf{st}' = \mathbf{st} \mid rs : aPV0) \vee$
$(\mathbf{st}(rs) = par1 \wedge \mathbf{st}(rw) = 0 \wedge \mathbf{st}' = \mathbf{st} \mid rs : aPV1)) \vee$
$(\mathbf{st}(rs) = par2 \wedge \mathbf{st}(rw) = 0 \wedge \mathbf{st}' = \mathbf{st} \mid rs : resg))$
The synchroniser chooses the Vr branch.

7.3 $\mathbf{e} = Vw{\uparrow} \wedge$
$((\mathbf{st}(rs) = gar0 \wedge \mathbf{st}(rw) = 0 \wedge \mathbf{st}(rr) = 0 \wedge \mathbf{st}' = \mathbf{st} \mid rs : aPV0) \vee$
$(\mathbf{st}(rs) = par1 \wedge \mathbf{st}(rw) = 0 \wedge \mathbf{st}(rr) = 0 \wedge \mathbf{st}' = \mathbf{st} \mid rs : aPV1) \vee$
$(\mathbf{st}(rs) = par2 \wedge \mathbf{st}(rw) = 0 \wedge \mathbf{st}(rr) = 0 \wedge \mathbf{st}' = \mathbf{st} \mid rs : resg))$
The synchroniser chooses the Vw branch.

8-9 same as $TRrn1$.

5. **Validity Conditions:**
The synchroniser does not generate anymore sequences that deadlock inside a PV-segment. But it can generate sequences that deadlock outside a PV-segment. One such sequence is for instance the following one:
The synchroniser is in the initial state. Suppose a component $rn2$ enters its first PV-segment. If this component leaves the first PV-segment it can choose non-deterministically between Vr, Vw or Vm events as last event of this segment. It can for instance always choose Vm. Suppose this is the case. It then eventually is added to the waiting set of semaphore r (see the specification of the binary semaphore) because it wants to enter its second PV-segment. Another component $rn2$ (or $wn2$) can now enter its first PV-segment because the first one chose the Vm event. Suppose this happens and this component chooses also Vm as its last event of the first PV-segment. This can continue until all $rn2$ components are in

the waiting set of semaphore r and all the $wn2$ components are in the waiting set of semaphore w. There is no component that can get these components out of the waiting sets, i.e., the system has deadlocked.
Hence, we must restrict $rn2$ and $wn2$ in such a way that they choose the right V-branch when they leave a PV-segment. In case of the example, the right V-branch for the last component is not Vm because there are no components that can generate tPm events as first $tP\ldots$ event on a semaphore. The last component must choose a Vr or Vw event. $Vrn2$ must express what the allowed sequences of $rn2$ are, i.e., not a deadlocked one as mentioned above.
$Vrn2 \equiv Rrn2 \rightarrow Grn2$
$Rrn2 \equiv$

- $\Box(\mathbf{st}(rs) = rung \rightarrow \Diamond(\mathbf{st}(rs) = bPV2))$
  $rn2$ relies on its reader to guarantee that its execution of `READ` only takes a finite amount of time.
- $\wedge\Box(\mathbf{e} = tPm \rightarrow \Diamond(\mathbf{e} = ePm))$
  $rn2$ furthermore relies on m that it gives eventually the access-right to the shared variables of the first or last PV-segment.
- $\wedge\Box(\mathbf{e} = tPr \rightarrow \Diamond(\mathbf{e} = ePr))$
  $rn2$ furthermore relies on r that it gives eventually the access-right to the shared variables of the second PV-segment.

$Grn2 \equiv$

- $\Box(\mathbf{st}(rs) = tryg \rightarrow \Diamond(\mathbf{st}(rs) = rung))$
  $rn2$ guarantees to its reader that it eventually may execute `READ` if he has requested it.
- $\wedge\Box(\mathbf{e} = ePm \rightarrow \Diamond(\mathbf{e} = Vm \wedge X1))$
  $rn2$ guarantees that it has the access-right in the first or third PV-segment only a finite amount of time and chooses the Vm-branch if the number of processes that have as first coming P-operation a P-operation on `m`, is greater than zero.
- $\wedge\Box(\mathbf{e} = ePr \rightarrow \Diamond(\mathbf{e} = Vm \wedge X1))$
  $rn2$ guarantees that it has the access-right in the second PV-segment only a finite amount of time and chooses the Vm-branch if the number of processes that have as first coming P-operation a P-operation on `m`, is greater than zero.
- $\wedge\Box(\mathbf{e} = ePm \rightarrow \Diamond(\mathbf{e} = Vr \wedge X2))$
  $rn2$ guarantees that it has the access-right in the first or third PV-segment only a finite amount of time and chooses the Vr-branch if the number of processes that have as first coming P-operation a P-operation on `r`, is greater than zero.
- $\wedge\Box(\mathbf{e} = ePr \rightarrow \Diamond(\mathbf{e} = Vr \wedge X2))$
  $rn2$ guarantees that it has the access-right in the second PV-segment

only a finite amount of time and chooses the Vr-branch if the number of processes that have as first coming P-operation a P-operation on `r`, is greater than zero.

- $\wedge\Box(\mathbf{e} = ePm \rightarrow \Diamond(\mathbf{e} = Vw \wedge X3))$
  $rn2$ guarantees that it has the access-right in the first or third PV-segment only a finite amount of time and chooses the Vw-branch if the number of processes that have as first coming P-operation a P-operation on `w`, is greater than zero.
- $\wedge\Box(\mathbf{e} = ePr \rightarrow \Diamond(\mathbf{e} = Vw \wedge X3))$
  $rn2$ guarantees that it has the access-right in the second PV-segment only a finite amount of time and chooses the Vw-branch if the number of processes that have as first coming P-operation a P-operation on `w`, is greater than zero.

$X1 \equiv \sum$ "components that from now on can generate a tPm event as first $tP\ldots$ event on a semaphore". > 0
$X2 \equiv \sum$ "components that from now on can generate a tPr event as first $tP\ldots$ event on a semaphore". > 0
$X3 \equiv \sum$ "components that from now on can generate a tPw event as first $tP\ldots$ event on a semaphore". > 0

3.4 The third development step

Dijkstra's solution to the problem of the newly introduced deadlocked sequences is as follows: record in a shared variable bX the number of components that can generate a P-operation on a semaphore X as their first coming P-operation. A component that executed a P-operation on X decreases bX by one. The component "knows" what its next P-operation is, so it increases the corresponding shared variable by one. The guards in the `CHOOSE` segment are changed so that the correct V-branch is chosen. The initial value of bm is $N + M$ because initially all processes have `P(m)` as their first coming P-operation. The initial value of br and bw is then of course 0. Like in the second step the initial value of m is 1 and that of ar, aw, r and w 0. The result of this transformation is as follows:

```
reader3:
          do true → NCS;
                    P(m);bm:=bm-1;br:=br+1;CHOOSE;
                    P(r);br:=br-1;ar:=ar+1;bm:=bm+1;CHOOSE;
                    READ;
                    P(m);bm:=bm-1;ar:=ar-1;bm:=bm+1;CHOOSE
          od

writer3:
          do true → NCS;
```

```
                    P(m);bm:=bm-1;bw:=bw+1;CHOOSE;
                    P(w);bw:=bw-1;aw:=aw+1;bm:=bm+1;CHOOSE;
                    WRITE;
                    P(m);bm:=bm-1;aw:=aw-1;bm:=bm+1;CHOOSE
          od
```

with `CHOOSE`: **if** `bm>0` $\rightarrow$`V(m)`
$[]$ `aw=0` $\wedge$ `br>0` $\rightarrow$`V(r)`
$[]$ `aw=0` $\wedge$ `ar=0` $\wedge$ `bw>0` $\rightarrow$`V(w)`
fi

`S3` : $\|_{i=1}^{N}$ `reader3` $\|$ $\|_{j=1}^{M}$ `writer3`

`S3` still generates sequences that Dijkstra does not allow. These sequences are generated because `CHOOSE` is still non-deterministic. Suppose a `reader3` can choose between a `V(m)` and a `V(r)` operation. Choosing `V(m)` causes that another `reader3` (`writer3`) can signal that it has finished executing `READ` (`WRITE`) or wants to execute `READ` (`WRITE`). A `V(r)` causes that a `reader3` can execute `READ`. Choosing `V(m)` thus unnecessarily blocks a `reader3`. So it is not a deadlocked sequence but only an inefficient sequence. The informal requirement of `S3` is that no unnecessary blocking sequences are allowed. Again, not a pure liveness requirement is added.

3.4.1 Specification of components $rn3$

For the synchroniser this means that $rn2$ and $wn2$ have to be changed and components bm, rm and wm have to be added. The changes to $rn2$ result in $rn3$:(Note: again the numbers of the transitions of the second implementation correspond with those of the following third implementation.)

1. **Events:**
   $Ern3 = Ern2 \bigcup \{gbm(x)\uparrow, gbw(z)\uparrow, gbr(y)\uparrow\ , pbm(x)\uparrow, pbr(y)\uparrow\colon x, y, z \in N\}$

2. **States:**
   $Qrn3 : rs : RS3 \times rr : N \times rw : N \times bm : N \times br : N \times bw : N$
   where
   $RS3 = RS2 \bigcup \{gbm0, gbr0, gbw0, pbm0, pbr0, gbm1, gbr1, gbw1, pbr1, pbm1, gbm2, gbr2, gbw2, pbm2, pbm3\}$

3. **Initial States**
   $IQrn3 \equiv \mathbf{st}(rs) = resg$

4. **Transitions:**
   $TRrn3 \equiv$

   1-5 same as $TRrn2$.

6.1 $\mathbf{e} = gbm(x)\!\uparrow \wedge((\mathbf{st}(rs) = gar0 \wedge \mathbf{st}' = \mathbf{st} \mid rs, bm : gbm0, x)\vee$
$(\mathbf{st}(rs) = gar1 \wedge \mathbf{st}' = \mathbf{st} \mid rs, bm : gbm1, x)\vee$
$(\mathbf{st}(rs) = gar2 \wedge \mathbf{st}' = \mathbf{st} \mid rs, bm : gbm2, x))$
The synchroniser retrieves the current value of bm.

6.2 $\mathbf{e} = gbr(y)\!\uparrow \wedge((\mathbf{st}(rs) = gbm0 \wedge \mathbf{st}' = \mathbf{st} \mid rs, br : gbr0, y)\vee$
$(\mathbf{st}(rs) = gbm1 \wedge \mathbf{st}' = \mathbf{st} \mid rs, br : gbr1, y)\vee$
$(\mathbf{st}(rs) = gbm2 \wedge \mathbf{st}' = \mathbf{st} \mid rs, br : gbr2, y))$
The synchroniser retrieves the current value of br.

6.3 $\mathbf{e} = gbw(z)\!\uparrow \wedge((\mathbf{st}(rs) = gbr0 \wedge \mathbf{st}' = \mathbf{st} \mid rs, bwgbw0, z)\vee$
$(\mathbf{st}(rs) = gbr1 \wedge \mathbf{st}' = \mathbf{st} \mid rs, bw : gbw1, z)\vee$
$(\mathbf{st}(rs) = gbr2 \wedge \mathbf{st}' = \mathbf{st} \mid rs, bwgbw2, z))$
The synchroniser retrieves the current value of bw.

6.4 $\mathbf{e} = pbm(x)\!\uparrow \wedge$
$((x = \mathbf{st}(bm) - 1 \wedge \mathbf{st}(rs) = gbw0 \wedge \mathbf{st}' = \mathbf{st} \mid rs, bm : pbm0, x)\vee$
$(x = \mathbf{st}(bm) + 1 \wedge \mathbf{st}(rs) = par1 \wedge \mathbf{st}' = \mathbf{st} \mid rs, bm : pbm1, x)\vee$
$(x = \mathbf{st}(bm) - 1 \wedge \mathbf{st}(ns) = gbw2 \wedge \mathbf{st}' = \mathbf{st} \mid rs, bm : pbm2, x)\vee$
$(x = \mathbf{st}(bm) + 1 \wedge \mathbf{st}(rs) = par2 \wedge \mathbf{st}' = \mathbf{st} \mid rs, bm : pbm3, x))$
The synchroniser increases or decreases bm.

6.5 $\mathbf{e} = pbr(y)\!\uparrow \wedge$
$((y = \mathbf{st}(br) + 1 \wedge \mathbf{st}(rs) = pbm0 \wedge \mathbf{st}' = \mathbf{st} \mid rs, br : pbr0, y)\vee$
$(y = \mathbf{st}(br) - 1 \wedge \mathbf{st}(rs) = gbw1 \wedge \mathbf{st}' = \mathbf{st} \mid rs, br : pbr1, y))$
The synchroniser increases or decreases br.

6.6 $\mathbf{e} = par(u)\!\uparrow \wedge$
$((u = \mathbf{st}(rr) + 1 \wedge \mathbf{st}(rs) = pbr1 \wedge \mathbf{st}' = \mathbf{st} \mid rs, rr : par1, u)\vee$
$(u = \mathbf{st}(rr) - 1 \wedge \mathbf{st}(rs) = pbm2 \wedge \mathbf{st}' = \mathbf{st} \mid rs, rr : par2, u))$
The synchroniser increases or decreases ar.

7.1 $\mathbf{e} = Vm\!\uparrow \wedge((\mathbf{st}(rs) = pbr0 \wedge \mathbf{st}(bm) > 0 \wedge \mathbf{st}' = \mathbf{st} \mid rs : aPV0)\vee$
$(\mathbf{st}(rs) = pbm1 \wedge \mathbf{st}(bm) > 0 \wedge \mathbf{st}' = \mathbf{st} \mid rs : aPV1)\vee$
$(\mathbf{st}(rs) = pbm3 \wedge \mathbf{st}(bm) > 0 \wedge \mathbf{st}' = \mathbf{st} \mid rs : resg))$
The synchroniser chooses the Vm branch.

7.2 $\mathbf{e} = Vr\!\uparrow \wedge$
$((\mathbf{st}(rs) = pbr0 \wedge \mathbf{st}(rw) = 0 \wedge \mathbf{st}(br) > 0 \wedge \mathbf{st}' = \mathbf{st} \mid rs : aPV0)\vee$
$(\mathbf{st}(rs) = pbm1 \wedge \mathbf{st}(rw) = 0 \wedge \mathbf{st}(br) > 0 \wedge \mathbf{st}' = \mathbf{st} \mid rs : aPV1)\vee$
$(\mathbf{st}(rs) = pbm3 \wedge \mathbf{st}(rw) = 0 \wedge \mathbf{st}(br) > 0 \wedge \mathbf{st}' = \mathbf{st} \mid rs : resg))$
The synchroniser chooses the Vr branch.

7.3 $\mathbf{e} = Vw\!\uparrow \wedge$
$((\mathbf{st}(rs) = pbr0 \wedge \mathbf{st}(rw) = 0 \wedge \mathbf{st}(rr) = 0\wedge$
$\mathbf{st}(bw) > 0 \wedge \mathbf{st}' = \mathbf{st} \mid rs : aPV0)\vee$
$(\mathbf{st}(rs) = pbm1 \wedge \mathbf{st}(rw) = 0 \wedge \mathbf{st}(rr) = 0\wedge$
$\mathbf{st}(bw) > 0 \wedge \mathbf{st}' = \mathbf{st} \mid rs : aPV1)\vee$
$(\mathbf{st}(rs) = pbm3 \wedge \mathbf{st}(rw) = 0 \wedge \mathbf{st}(rr) = 0\wedge$
$\mathbf{st}(bw) > 0 \wedge \mathbf{st}' = \mathbf{st} \mid rs : resg))$
The synchroniser chooses the Vw branch.

8-9 same as $TRrn2$.

5. **Validity Conditions:**
   Dijkstra gives priority to a Vr (Vw) event if it is possible to choose between Vr (Vw) and Vm. This informal requirement we formalise by $Vrn3$.
   $Vrn3 \equiv Rrn3 \rightarrow Grn3$
   $Rrn3 \equiv$

   - $\Box(\mathbf{st}(rs) = rung \rightarrow \Diamond(\mathbf{st}(rs) = bPV2))$
     $rn3$ relies on its reader to guarantee that its execution of `READ` only takes a finite amount of time.
   - $\wedge\Box(\mathbf{e} = tPm \rightarrow \Diamond(\mathbf{e} = ePm))$
     $rn3$ furthermore relies on m that it gives eventually the access-right to the shared variables of the first or last PV-segment.
   - $\wedge\Box(\mathbf{e} = tPr \rightarrow \Diamond(\mathbf{e} = rePr))$
     $rn3$ furthermore relies on r that it gives eventually the access-right to the shared variables of the second PV-segment.

   $Grn3 \equiv$

   - $\Box(\mathbf{st}(rs) = tryg \rightarrow \Diamond(\mathbf{st}(rs) = rung))$
     $rn3$ guarantees to its reader that it eventually may execute `READ` if he has requested it.
   - $\wedge\Box(\mathbf{e} = ePm \rightarrow \Diamond(\mathbf{e} = Vm \wedge X1))$
     $rn3$ guarantees that it only takes the Vm-branch if the Vr- and Vw-branch can not be taken.
   - $\wedge\Box(\mathbf{e} = ePr \rightarrow \Diamond(\mathbf{e} = Vm \wedge X1))$
     $rn3$ guarantees that it only takes the Vm-branch if the Vr- and Vw-branch can not be taken.
   - $\wedge\Box(\mathbf{e} = ePm \rightarrow \Diamond(\mathbf{e} = Vw \wedge X2))$
     $rn3$ guarantees that it only takes the Vw-branch if the Vm-branch can not be taken.
   - $\wedge\Box(\mathbf{e} = ePr \rightarrow \Diamond(\mathbf{e} = Vw \wedge X2))$
     $rn3$ guarantees that it only takes the Vw-branch if the Vm-branch can not be taken.
   - $\wedge\Box(\mathbf{e} = ePm \rightarrow \Diamond(\mathbf{e} = Vr \wedge X3))$
     $rn3$ guarantees that it only takes the Vr-branch if the Vm-branch can not be taken.
   - $\wedge\Box(\mathbf{e} = ePr \rightarrow \Diamond(\mathbf{e} = Vr \wedge X3))$,
     $rn3$ guarantees that it only takes the Vr-branch if the Vm-branch can not be taken.

   where

- $X1 \equiv (\mathbf{st}(bm) > 0 \wedge \neg X2 \wedge \neg X3))$
  is the condition for the Vm-branch and
- $X2 \equiv (\mathbf{st}(rw) = 0 \wedge \mathbf{st}(br) > 0)$
  is the condition for the Vr-branch and
- $X3 \equiv (\mathbf{st}(rw) = 0 \wedge \mathbf{st}(rr) = 0 \wedge \mathbf{st}(bw) > 0)$
  the condition for the Vw-branch.

3.5 The fourth development step

We have already seen how we can prevent $rn3$ to choose wrongly between Vr and Vw. Dijkstra also updates the PV-segments in such a way that only statements that are actually executed are listed. It turns out that we do not anymore need `bm`. Also the guards of `CHOOSE` get simpler. The result of this transformation is:

```
reader4:
  do true → NCS;
            P(m);br:=br+1;if aw>0 → V(m) [] aw=0 →V(r) fi;
            P(r);br,ar:=br-1,ar+1;if br=0 → V(m) [] br>0 →V(r)fi;
            READ;
            P(m);ar:=ar-1;
            if ar>0 ∨ bw=0 →V(m) [] ar=0 ∧ bw>0 → V(w) fi
  od

writer4:
  do true → NCS;
            P(m);bw:=bw+1;
            if aw>0 ∨ ar>0 → V(m) [] aw=0 ∧ ar=0 → V(w) fi;
            P(w);bw,aw:=bw-1,aw+1;V(m);
            WRITE;
            P(m);aw:=aw-1;
            if br=0 ∧ bw=0 → V(m) [] br>0 → V(r) [] bw>0 → V(w)fi
  od
```

`S4 :` $\|_{i=1}^{N}$ `reader4` $\|$ $\|_{j=1}^{M}$ `writer4`

In `S4` only the last `CHOOSE` operation of `writer4` is non-deterministic, i.e. there is a choice between a `V(r)` and a `V(w)` operation. Dijkstra suggests to give priority to `V(r)`.

3.5.1 Specification of components $Mrn4$

In Stark's formalism these modifications leads to specification
$Srn4 = \langle Mrn4, Vrn4 \rangle$ ($Swn4$), as specified below:

1. **Events:**
   $Ern4 = Ern3 \setminus \{gbm(x)\!\uparrow, pbm(x)\!\uparrow : x \in N\}$

2. **States:**
$Qrn4 : rs : RS4 \times rr : N \times rw : N \times br : N \times bw : N$
where
$RS4 = RS3 \setminus \{gaw1, gar0, gaw2, gbm0, gbw0, pbm0, gbm1, pbm1, gbm2, gbr2, pbm2, pbm3\}$

3. **Initial States:**
$IQrn4 \equiv \mathbf{st}(rs) = resg$

4. **Transitions:**
$TRrn4 \equiv$

1-3 same as $TRrn3$.

4 $\mathbf{e} = gaw(w)\uparrow \wedge \mathbf{st}(rs) = iPV0 \wedge \mathbf{st}' = \mathbf{st} \mid rs, rw : gaw0, w$
We need the value of aw only in the first PV-segment.

5 $\mathbf{e} = gar(v)\uparrow \wedge((\mathbf{st}(rs) = iPV1 \wedge \mathbf{st}' = \mathbf{st} \mid rs, rr : gar1, v)\vee$
$(\mathbf{st}(rs) = iPV2 \wedge \mathbf{st}' = \mathbf{st} \mid rs, rr : gar2, v))$
We need the value of ar only in the second and third PV-segment.

6.1 $\mathbf{e} = gbr(y)\uparrow \wedge((\mathbf{st}(rs) = gaw0 \wedge \mathbf{st}' = \mathbf{st} \mid rs, br : gbr0, y)\vee$
$(\mathbf{st}(rs) = gar1 \wedge \mathbf{st}' = \mathbf{st} \mid rs, br : gbr1, y))$
We need the value of br only in the first and second PV-segment.

6.2 $\mathbf{e} = gbw(z)\uparrow \wedge \mathbf{st}(rs) = gar2 \wedge \mathbf{st}' = \mathbf{st} \mid rs, bw : gbw2, z$
We need the value of bw only in the third PV-segment.

6.3 $\mathbf{e} = pbr(y)\uparrow \wedge$
$((y = \mathbf{st}(br) + 1 \wedge \mathbf{st}(rs) = gbr0 \wedge \mathbf{st}' = \mathbf{st} \mid rs, br : pbr0, y)\vee$
$(y = \mathbf{st}(br) - 1 \wedge \mathbf{st}(rs) = gbr1 \wedge \mathbf{st}' = \mathbf{st} \mid rs, br : pbr1, y))$
The value of br is updated only in the first and second PV-segment.

6.4 $\mathbf{e} = par(u)\uparrow \wedge$
$((u = \mathbf{st}(rr) + 1 \wedge \mathbf{st}(rs) = pbr1 \wedge \mathbf{st}' = \mathbf{st} \mid rs, rr : par1, u)\vee$
$(u = \mathbf{st}(rr) - 1 \wedge \mathbf{st}(rs) = gbw2 \wedge \mathbf{st}' = \mathbf{st} \mid rs, rr : par2, u))$
The value of ar is updated only in the second and third PV-segment.

7.1 $\mathbf{e} = Vm\uparrow \wedge$
$((\mathbf{st}(rs) = pbr0 \wedge \mathbf{st}(rw) > 0 \wedge \mathbf{st}' = \mathbf{st} \mid rs : aPV0)\vee$
$(\mathbf{st}(rs) = par1 \wedge \mathbf{st}(br) = 0 \wedge \mathbf{st}' = \mathbf{st} \mid rs : aPV1)\vee$
$(\mathbf{st}(rs) = par2 \wedge (\mathbf{st}(rr) > 0 \vee \mathbf{st}(bw) = 0) \wedge \mathbf{st}' = \mathbf{st} \mid rs : resg))$
The guards of Vm simplify to this.

7.2 $\mathbf{e} = Vr\uparrow \wedge((\mathbf{st}(rs) = pbr0 \wedge \mathbf{st}(rw) = 0 \wedge \mathbf{st}' = \mathbf{st} \mid rs : aPV0)\vee$
$(\mathbf{st}(rs) = par1 \wedge \mathbf{st}(br) > 0 \wedge \mathbf{st}' = \mathbf{st} \mid rs : aPV1))$
The guards of Vr simplify to this.

7.3 $\mathbf{e} = Vw\uparrow \wedge \mathbf{st}(rs) = par2 \wedge \mathbf{st}(bw) = 0\wedge$
$\mathbf{st}(rr) = 0 \wedge \mathbf{st}' = \mathbf{st} \mid rs : resg$
The guards of Vw simplify to this.

8-9 same as $TRrn3$

5. **Validity Conditions:**
$Vrn4 \equiv Rrn4 \rightarrow Grn4$
$Rrn4 \equiv \Box(\mathbf{st}(rs) = rung \rightarrow \Diamond(\mathbf{st}(rs) = bPV2))\wedge$
$\Box(\mathbf{e} = tPm \rightarrow \Diamond(\mathbf{e} = ePm))\wedge$
$\Box(\mathbf{e} = tPr \rightarrow \Diamond(\mathbf{e} = ePr))$
$Grn4 \equiv \Box(\mathbf{st}(rs) = tryg \rightarrow \Diamond(\mathbf{st}(rs) = rung))\wedge$
$\Box(\mathbf{e} = ePm \rightarrow \Diamond(\mathbf{e} = Vm))\wedge$
$\Box(\mathbf{e} = ePr \rightarrow \Diamond(\mathbf{e} = Vm))\wedge$
$\Box(\mathbf{e} = ePm \rightarrow \Diamond(\mathbf{e} = Vr))\wedge$
$\Box(\mathbf{e} = ePr \rightarrow \Diamond(\mathbf{e} = Vr))\wedge$
$\Box(\mathbf{e} = ePm \rightarrow \Diamond(\mathbf{e} = Vw))\wedge$
$\Box(\mathbf{e} = ePr \rightarrow \Diamond(\mathbf{e} = Vw))$

The requirement of giving priority to a `V(r)` operation is easily formulated, so we do not give it. Note that, formally speaking, we have not finished the implementation since we did not implement the abstract semaphore. Since doing so is not difficult, we leave it at that (take an implementation for a strong semaphore and transform it into this formalism).

4 Conclusion

We have shown by the formal development of Dijkstra's readers/writers program that it is indeed possible to formalise Dijkstra's development strategy of deleting undesirable sequences generated by intermediate programs. We have formalised this development within Stark's formalism which expresses separately the safety and liveness properties of a program under development. We offer the following conclusions:

- The formalisation of liveness causes no problems; we can translate the liveness conditions required for shared variables and semaphores into Stark's formalism.
- The translation of high level liveness properties into low level safety and liveness properties also causes no problems.
- The main problem is that this translation sometimes generates new, not allowed, sequences that on a higher level were previously not possible. This problem is solved by disallowing such sequences with the help of validity conditions which remove disallowed sequences from a machine. These validity conditions were originally intended in Stark's formalism to describe the liveness conditions. We have used them for *another purpose*: to extract the allowed sequences of a machine.

The last observation implies that a notion of satisfaction that uses set inclusion between sets of sequences generated by the safety parts only is not the correct one. We would nevertheless like to preserve Dijkstra's treatment of the readers/writers problem in this formalisation. From the example in the paper it can be seen how we achieve this. The proof obligation on the machine parts abstracts away differences that are caused by potential deadlock or blocking. Namely by the combined use of stutter steps as well as abstraction functions. That this does not cause inconsistencies is because of the limited corrective role of the validity condition: any liveness properties that a high level machine enables should also belong to the potential of the low level one. Only potential deadlock or blocking can be corrected. This turns out to be exactly the kind of incorrect sequences that Dijkstra allows in his approximative development. This means that the direction of the development is not only from validity conditions to machines but also from machines to validity conditions.

In future work we want to also apply Stark's formalism to the development of fault tolerant systems. Also the formalism should be changed so that machine specifications get shorter.

4.1 Acknowledgements

I would like to thank W.H.J. Feijen for drawing our attention to [2] as an example of developing a parallel program.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. In *Third annual symposium on Logic in Computer Science*, pages 165–175, July 1988.

[2] E.W. Dijkstra. A tutorial on the split binary semaphore, 1979. EWD 703.

[3] E. Diepstraten and R. Kuiper. Abadi & Lamport and Stark: towards a proof theory for stuttering, dense domains and refinements mappings. In *LNCS 430:Proc. of the REX Workshop on Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness*, pages 208–238. Springer-Verlag, 1990.

[4] C.B. Jones. *Development methods for computer programs including a notion of interference.* PhD thesis, Oxford University Computing Laboratory, 1981.

[5] L. Lamport. What good is temporal logic. In R.E.A. Manson, editor, *Information Processing 83: Proc. of the IFIP 9th World Congress*, pages 657–668. Elsevier Science Publishers, North Holland, 1983.

[6] L. Lamport. An axiomatic semantics of concurrent programming languages. In K.R. Apt, editor, *NATO ASI SERIES, vol. F13: Logics and Models of Concurrent Systems*, pages 77–122. Springer-Verlag, January 1985.

[7] L. Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, January 1989.

[8] S. Lee, S. Gerhart, and W.-P. de Roever. The evolution of list-copying algorithms and the need for structured program verification. In *Proc. of 6th POPL*, 1979.

[9] J. Misra, and M. Chandy. Proofs of Networks of Processes. IEEE SE 7 (4), pp. 417-426, 1981.

[10] R. Milner. A calculus of Communicating Systems. LNCS 92, Springer-Verlag 1980.

[11] P.R.H. Place, W.G. Wood, and M. Tudball. Survey of formal specification techniques for reactive systems. Technical Report, 1990.

[12] E.W. Stark. *Foundations of a Theory of Specification for Distributed Systems*. PhD thesis, Massachusetts Inst. of Technology, 1984. Available as Report No. MIT/LCS/TR-342.

[13] E.W. Stark. A Proof Technique for Rely/Guarantee Properties. In *LNCS 206: Fifth Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 369–391. Springer-Verlag, 1985.

[14] E.W. Stark. Proving entailment between conceptual state specifications. *Theoretical Computer Science*, 56:135–154, 1988.

[15] J. Zwiers, A. de Bruin, and W.-P. de Roever. A proof system for partial correctness of Dynamic Networks of Processes. In proc. of the conference on logics of programs 1983, LNCS 164, Springer Verlag 1984.