The Refinement of Real-Time Systems*

D. J. Scholefield H.S.M. Zedan
Formal Systems Research Group
Department of Computer Science
University of York. York. YO1 5DD. UK

Abstract

The Temporal Agent Model (TAM) is a wide-spectrum development language for real-time systems. In TAM, limited resources are modelled by deriving release times and absolute deadlines from weakest pre-condition predicate transformers. In this paper the language syntax is described, a refinement calculus is presented and a number of examples are discussed.

1 Introduction

The formal development of a real-time system can be divided into three separate tasks; requirements specification, concrete design, and verification between the former two activities. In real-time systems, verification must take into account not only functional correctness but also the timeliness of results.

In many real-time formal theories, two independent languages are used for the specification and design tasks. For example, in Timed-CSP [Dav91] [Schn90] and Timed-CCS [TM89] [Tof90] first-order predicate logic is used as the requirements specification language, and a process algebra defines the concrete design language. Verification is undertaken by proving that the logic formulae satisfy the traces arising from the processes.

However, a problem inherent in such 'two tiered' theories is the lack of method by which suitable designs are arrived at. A combination of experience and guess-work must be used in order to develop processes, and then verification – a rather time consuming activity – is undertaken. If the verification fails, then the processes are re-formulated and verification is undertaken again. There are some methods which overcome this specific problem, Ostroff [Ost89] has devised a synthesis method which enables the user to derive designs from specifications, however the method is conservative (it does not always find a suitable design even when one may exist). Other methods suffer from similar drawbacks.

To overcome this problem we have devised the Temporal Agent Model (TAM) [SZ91] [SZ92] [Sch92], which is a theory centered upon a 'wide- spectrum' language in which both real-time requirements specifications and concrete designs may be formulated. A specification in TAM may be transformed step-by-step into a mixed program containing both specification statements and concrete design statements. Such transformations continue until a completely concrete program is produced which is guaranteed correct with respect to the original specification. The transformation process is known widely as refinement, and is already common in purely transformational systems [Mor90] [MRG88] [SETL85] [CIP85] [BW90].

TAM uses conservative extensions to first-order predicate logic to express the time at which variables hold certain values. The logic is used both as a basis for a language statement, and to define the timed predicate transformers for the language's semantics. The concrete language statements are loosely based on Dijkstra's guarded command language [Dij76] with real-time extensions to cover deadlines, delays and concurrency.

2 A Computational Model

We define a **real-time system** as a collection of concurrently executing computation **agents** which communicate asynchronously via time-stamped shared data areas called **shunts**. Systems can themselves be viewed as single agents and composed into larger systems. Systems may have timing constraints imposed at three different levels; system-wide communication deadlines, agent deadlines, (both on the agent computations and inter-agent communication) and sub-computation deadlines (within the computation of an individual agent). A system has a static configuration, i.e. all connections remain fixed throughout the lifetime of the system. Every system also has the constraint that there may be at most a finite number of shunts and agents.

Time is global i.e. a single clock is accessible to every agent and shunt. It is both discrete and linear, and modelled naturally by the positive integers. There is a unique 'first time' instant from which we assume all systems will measure their execution release times for agents, and an upper bound (denoted ∞) which is greater than, or equal to, any time.

An agent is described by a computation which may transform a private data-space and may read and write shunts during execution. The computation may be non-deterministic, and have both minimum and maximum execution times imposed. In particular, agents have the following properties:

- Every agent must terminate,

*This work is partially supported by SERC grant GR/F 35920/4/1/1214

- Only an agent which performs no computation or reading or writing may be instantaneous (i.e. terminate at the same instant in time as it is released).
- An agent may start execution as a result of either a condition on the current time, or a write event occurring on a specific shunt. These two conditions model periodic and sporadic real-time tasks respectively.
- An agent may have deadlines on computation, sub-computations, and communication reads and writes. Deadlines may be **dynamic** i.e. dependent upon results of computation, or **static**. Deadlines are all considered to be **hard**, that is, there is no concept of deadline priority.
- The size of any agent's data-space is finite.
- An agent may write to at most a finite number of shunts, and read from at most a finite number of shunts.
- An agent's output shunts are *owned* by that agent, i.e. no other agent may write to those shunts (although many other agents may read them).

Shunts are passive shared memory spaces which contain two values; the first gives the time at which the most recent write took place, and the second gives the value which was most recently written. Shunt writing is destructive, shunt reading is not. Shunts are strictly typed throughout the lifetime of the system, i.e. there is an identified set of values to which the data value of a shunt must always belong.

Shunts are loosely asynchronous; if the owning agent performs a write then the write is said to be completed when the value has been accepted by the shunt and the time-stamp is over-written by the current time. The only delay on writing is due to the communication overhead, a shunt may not refuse to accept value. When an agent performs a read then the shunt returns both the time-stamp found in the shunt, and the associated value. Again, any delay is due to communication overheads.

2.1 A Logic to Describe Computations

We make conservative extensions to the first-order predicate logic, and import standard ZF set theory. The extensions equate computational variables and shunts with functions mapping positive integers (time) onto value sets.

A **timed variable** is a function mapping positive integers (representing the time) onto some value set (representing the variable type). Given a computation variable 'x' of type 'V' ($x \in [\mathcal{N} \rightarrow V]$), then we use the notation $x@t$ to denote the application of the function 'x' to the time 't', i.e.: $x@t \quad =_{def} \quad x(t)$.

This allows us to make assertions of the form (for example):

$$\forall n \in \mathcal{N}(x@(t+1) = x@t)$$

(the variable x has the same value for all times, i.e. is a *constant*).

Shunts are also represented by functions mapping positive integers (times) onto some value set. The value set will always be a cross-product of positive integers and some value set. Again, we use the notation $s@t$ to represent the value of the shunt 's' at some time 't'. The value of a shunt will always be a timestamp and value pair, so we will use the tuple accessors $.ts$ and $.v$ to access a shunt value, e.g.

$$s@t \;=\; ((s@t).ts \;,\; (s@t).v)$$

2.2 Timed Logic and Specifications

One of the syntactic program statements in the TAM language is the *specification statement*. In a specification statement $w : \Phi$, w lists those variables and shunts which may change during the execution of an agent which satisfies Φ. The set w is called a frame. The formula Φ may contain the free natural number variables t and t', representing the agent's release and termination times. Computation variables may only be referred to at times t and t', and shunts may only be referred to at times in $[t, t']$. The formula therefore represents a timed transition on computation variables and shunts.

It is assumed that the external configuration of a TAM system is static, and therefore the environment of an agent, i.e. the shunts which are read and written by that agent, and the local state space, are known prior to system development. Therefore, in most cases, the initial specification can be viewed as a description of a 'black-box' with a number of identified input and output channels and a local state space definition.

3 TAM Syntax

The modular abstraction mechanism in TAM is the *agent* which can be defined recursively by the following ten agent forms:

$x := e$ performs an assignment of the value of the expression e (defined over the local state space of the agent) to the variable x. The expression evaluation can never be instantaneous.

$x \leftarrow s$ performs an input from the shunt s into the variable x. The type of x must therefore be a natural number and value pair. The input is asynchronous and may never be instantaneous.

$x \rightarrow s$ performs an asynchronous output to the shunt s from the variable (or constant value) x. The time of the output is used as the time stamp. Output may not be instantaneous.

$w : \Phi$ is a specification statement over the variables in the local state space of the agent, and the shunts in the environment of the agent.

$(x)\mathcal{A}$ introduces the unique variable x into the local state space of the agent $\mathcal{A}$. Initially x has an undefined value (any value from the appropriate type set

T). The variable is destroyed (unavailable) after the termination of $\mathcal{A}$.

$\mathcal{A}/s$ restricts access to the shunt s to the agent $\mathcal{A}$, and declares it's type as T. This operator can be seen as a 'connector' of two ports in that no communication can actually take place until a shunt is restricted.

$\mathcal{A}|\mathcal{A}$ executes the two agents concurrently. The two agents are released at the same time, and the concurrent composition terminates some arbitrary time after both agents terminate. We use the shorthand $\prod_{i \in I} \mathcal{A}_i$ to denote indexed concurrent composition (I is finite).

$[t]\mathcal{A} : \mathcal{A}$ assumes that the execution of the first agent will terminate sometime before t time units after release. The second agent will be released at the time that the first agent terminates. This provides a constraint on the run-time execution environment (in particular, the scheduler).

$\bigsqcup_{i \in I} g_i \Rightarrow \mathcal{A}_i$ evaluates all of the guards g_i (which are boolean expressions on local state space and shunts), and executes one of the agents corresponding to a true guard. If no guards evaluate to true, then the choice agent terminates correctly. It is also assumed that a function Δ is pre-defined which defines the evaluation time necessary for any guard g_i. (The indexing set I is always finite).

$\mathcal{A} \triangleright^s_n \mathcal{A}$ is a timeout operator. The agent waits for a write to occur on shunt s for n time units. As soon as the shunt is written to the right-hand agent is released. If after n time units no write has occurred then the left-hand agent is released.

$\mu X_n.F(X)$ represents that agent which executes the agent F and upon encountering the agent variable X will start executing F again. This recursion is bounded by the natural number variable n (where $n < \infty$).

We impose the syntactic restriction that no agent may share its' local state space with a concurrently executing agent, and only one agent may write to any given shunt. The formal semantics for TAM are defined by extensions to Dijsktra's weakest pre-condition semantics [Dij76] and are described in full in [SZ92] [Sch92].

From the primitive agents above we may derive the following useful constructs.

$\text{skip} \ =_{def} \ \emptyset : \text{true}$

$\delta n \ =_{def} \ (\text{skip} \triangleright^s_n \text{skip})/s$

$\text{abort} \ =_{def} \ [0]\delta 1 : skip$

$\mathcal{A}; \mathcal{B} \ =_{def} \ [\infty]\mathcal{A} : \mathcal{B}$

$[t]\mathcal{A} \ =_{def} \ [t]\mathcal{A} : \text{skip}$ (non sequence deadline).

4 Refinement

A formal definition of the refinement relation is given in the appendix, but it can be informally described as follows:

> "An agent $\mathcal{A}$ is refined by $\mathcal{B}$ whenever the environmental constraints which are necessary for $\mathcal{A}$ to achieve a given behaviour are stronger than those for $\mathcal{B}$, and, the effects of $\mathcal{B}$ on the environment are stronger (more deterministic) than those of $\mathcal{A}$"

A subset of the refinement calculus for TAM is presented in the appendix and we only discuss a representative sample here. The most important refinement law is that of specification strengthening:

$w : \Phi \ \sqsubseteq \ w : \Phi'$

(if $\Phi' \Rrightarrow \Phi$ and $\Sigma(w : \Phi) = \Sigma(w : \Phi')$)

This law asserts that if a user asks for an agent which ensures the behaviour Φ, and we provide the stronger behaviour Φ' then we are fulfilling at least the behaviour required of us. The constraint 'Σ' on specifications is defined in [SZ92] [Sch92], and intuitively defines the minimum amount of time required for the execution of the specification in order to ensure that the specification is feasible.

We can see that given a specification:

$\{x\} : x@t' = (x@t) + 1 \ \vee \ x@t' = (x@t) - 1$

the requirements are that the variable x is either incremented or decremented. We may therefore refine this to:

$\{x\} : x@t' = (x@t) - 1$

Similarly, a specification of an input from shunt s to variable x within the time interval $t, t+5$:

$\{x\} : \bigvee_{\sigma \in [t, t+5]} (x@t' = s@\sigma) \ \wedge \ t' \geq t + 5$

Can be refined to the input specification which requires the input to occur between $t+1$ and $t+5$:

$\{x\} : \bigvee_{\sigma \in [t+1, t+5]} (x@t' = s@\sigma) \ \wedge \ t' \geq t + 5$

Refinements on choice constructs must take into account the time required in which to evaluate the boolean guards, therefore the rule often found in refinement systems of the form:

$$\bigsqcup_{i \in I} g_i \Rightarrow \mathcal{A}_i \ \sqcup \ \text{true} \Rightarrow \mathcal{B} \quad \sqsubseteq \quad \mathcal{B}$$

is amended to:

$$\bigsqcup_{i \in I} g_i \Rightarrow \mathcal{A}_i \ \sqcup \ \text{true} \Rightarrow \mathcal{B} \quad \sqsubseteq$$

$$[\,^{\Sigma}_{i \in I} \Delta(g_i) + \Delta(true)]\delta \,^{\Sigma}_{i \in I} \Delta(g_i) + \Delta(true) : \mathcal{B}$$

In which it is assumed that the execution of the agent $\mathcal{B}$ is delayed precisely by the amount of time taken by the evaluation of all of the guards (including the guard 'true').

If we were to remove the timing considerations of the choice construct (i.e. $\Delta(g) = 0$ for all guards g), then the former refinement law would be valid:

$$\begin{array}{l} \bigsqcup_{i \in I} g_i \Rightarrow \mathcal{A}_i \ \sqcup \ \text{true} \Rightarrow \mathcal{B} \\ \quad \sqsubseteq \quad [0]\delta 0 : \mathcal{B} \\ \quad \sqsubseteq \quad \mathcal{B} \end{array}$$

where the last refinement assumes the law:

$$[0]\delta 0 : \mathcal{A} \quad \sqsubseteq \quad \mathcal{A}$$

Concurrency can be introduced by dividing a specification conjunction into concurrent specifications. This may be achieved by the following law:

$$\bigcup_{i \in I} w_i : \bigwedge_{i \in I} \Phi_i \quad = \quad \prod_{i \in I} w_i : \Phi_i$$
if w_i are disjoint

Note the requirement that the individual specifications are interference free, i.e. the frames w_i must be disjoint.

In general, the division of a specification into concurrent specifications is the only place in which concurrency is introduced into the refinement process.

5 Example - A Single-Shot Monitor

The requirements are to derive an agent which will read an input shunt s_0 within five time units of release, and if the value which was read passes a boolean test '*ok*', then no action is taken, if however the value read does not pass the test then the constant value '*alarm*' is written to the output shunt s_1. In the following refinement, the proof numbers refer to the laws defined in appendix. We may formalise the requirements by the following two properties:

Property 1:

$$(\bigwedge_{\sigma \in [t, t+5]} \neg ok(s_0@\sigma)) \ \Rightarrow \ write(s_1, alarm, t+5, t')$$

"if none of the values in s_0 are 'ok' for the first five time units then the alarm must be written to s_1"

Property 2: $t' \geq t + 6$

"there must be five units of time available in which to perform the reading, and one time unit in which to either write if necessary".

We can now see that:

$\{s_1\} : property1 \ \wedge \ property2 \quad \sqsubseteq \quad Spec1$
(by 12.6)

Spec1 $=_{def}$

$$\begin{array}{l} \{s_1\} : \exists x(\exists m(m \in [t, min(t+5, t')] \\ \ \wedge \ \bigvee_{\sigma \in [t, m']} (x@m = s_0@\sigma) \wedge \\ \quad stable(s_1, t, m) \ \wedge \ m \geq t+1 \\ \ \wedge \ ((\neg ok(x@m) \ \wedge \ write(s_1, alarm, m, t')) \\ \ \vee \ (ok(x@m) \ \wedge \ stable(s_1, m, t'))) \\ \ \wedge \ t' \geq m + 5)) \end{array}$$

In refining to *Spec*1 we are making a number of design decisions, but none of which contradict the informal specification.

We may now continue with the refinement process.

$$Spec1 \quad \sqsubseteq$$

$$\begin{array}{l} (x)\{s_1, x\} : \exists m(m \in [t, min(t+5, t')] \\ \ \wedge \ \bigvee_{\sigma \in [t, m']} (x@m = s_0@\sigma) \wedge \\ \quad stable(s_1, t, m) \ \wedge \ m \geq t+1 \\ \ \wedge \ ((\neg ok(x@m) \ \wedge \ write(s_1, alarm, m, t')) \\ \ \vee \ (ok(x@m) \ \wedge \ stable(s_1, m, t'))) \\ \ \wedge \ t' \geq m + 5) \qquad \text{(by 9.4)} \end{array}$$

$$\sqsubseteq$$

$$\begin{array}{l} (x)([5](\{s_1, x\} : \bigvee_{\sigma \in [t, t']} (x@t' = s_0@\sigma) \wedge \\ \quad stable(s_1, t, t') \ \wedge \ t' \geq t+1) \\ : \ \{s_1, x\} : ((\neg ok(x@t) \ \wedge \ write(s_1, alarm, t, t')) \\ \ \vee \ (ok(x@t) \ \wedge \ stable(s_1, t, t'))) \\ \ \wedge t' \geq t + 5 \ ) \qquad \text{(by 4.5)} \end{array}$$

We now split the sequence into the LHS and RHS. The LHS is refined first:

$$\begin{array}{l} \{s_1, x\} : \bigvee_{\sigma \in [t, t']} (x@t' = s_0@\sigma) \wedge \\ \quad stable(s_1, t, t') \ \wedge \ t' \geq t+1 \end{array}$$

$$\sqsubseteq \quad \{x\} : \bigvee_{\sigma \in [t, t']} (x@t' = s_0@\sigma) \ \wedge \ t' \geq t+1 \qquad \text{(by 12.3)}$$

$$\sqsubseteq \quad \{x\} : \bigvee_{\sigma \in [t+1, t']} (x@t' = s_0@\sigma) \ \wedge \ t' \geq t+1$$
(by 12.6 and 12.4)

$$\sqsubseteq \quad x \leftarrow s_0 \qquad \text{(by 2.2)}$$

Now the RHS side is refined:

$$\begin{array}{l} \{s_1, x\} : ((\neg ok(x@t) \ \wedge \ write(s_1, alarm, t, t')) \\ \qquad \vee \ (ok(x@t) \ \wedge \ stable(s_1, t, t'))) \\ \qquad \wedge t' \geq t + 5 \end{array}$$

$$\sqsubseteq$$

$$\begin{array}{l} \{s_1, x\} : ((\neg ok(x@t) \\ \quad \wedge \ write(s_1, alarm, t+1, t') \ \wedge \ t' \geq (t+1) + 4) \\ \vee \ (ok(x@t) \\ \quad \wedge \ stable(s_1, t+1, t') \ \wedge \ t' \geq (t+1) + 4)) \\ \wedge \ all_stable(\{s_1\}, t, t+1) \qquad \text{(by 12.6)} \end{array}$$

$$\sqsubseteq$$

$$\begin{array}{l} \neg ok(x) \Rightarrow \\ \quad (\{s_1\} : write(s_1, alarm, t, t') \mid \emptyset : t' \geq t+4) \\ ok(x) \Rightarrow \\ \quad (\{s_1\} : stable(s, t, t') \mid \emptyset : t' \geq t+4) \\ \quad \text{(by 7.9, 5.5,12.6, 12.7, if } \Delta(ok(x)) + \Delta(\neg ok(x)) = 1) \end{array}$$

$$\sqsubseteq$$

$\neg ok(x) \Rightarrow (alarm \rightarrow s_1) \mid \delta 4)$
$ok(x) \Rightarrow \delta 4$
(by 3.2,11.2,12.3,5.3).

We may now place our sequence halves back into the original agent to give:

$$(x)([5](x \leftarrow s_0) : \ ok(x) \Rightarrow \delta 4 \\ \neg ok(x) \Rightarrow (alarm \rightarrow s_1 \mid \delta 4))$$

6 Conclusion

We have presented a wide-spectrum language and refinement calculus for the development of real-time systems. It is clear that developing such systems will remain a complex and time consuming task independent of the method employed, but we have demonstrated that a refinement method is both feasible and useful for hard real-time systems.

Our development method does not address issues such as schedulability and process allocation. Work by Hooman [Hoo91] and Joseph [MJ90] implies that useful results can be obtained by modelling resource issues in the development language – we will investigate this further in the near future. However, TAM provides a language in which real-time specifications can be refined to task sets and profiles (given by the deadlines embedded in the concrete code), and interfaces with existing scheduling theory and run-time execution environments.

Perhaps the most important factor in formal development is that of complexity. The refinement calculus of TAM has complex proof obligations, in the future, an automated theorem checker such as HOL [Gor88] may be used.

7 References

[BW90] R. Back, J. Wright, "Refinement Concepts Formalised in Higher Order Logic". BCS-FACS Vol 2. No. 3. 1990.

[CIP85] The CIP Language Group, "The Munich Project CIP. Vol1", LNCS-183. 1985.

[Dav91] J. Davies, "Specification and Proof in Real-Time Systems". Oxford University, Ph.D Thesis PRG-93. 1991.

[Dij76] E. Dijkstra, "A Discipline Of Programming". Prentice-Hall. 1976.

[Gor88] M. J. Gordon, "Mechanizing Programming Logics in Higher-Order Logic", University of Cambridge Technical Report 145. September 1988.

[Hoo91] J. Hooman, "Specification and Compositional Verification of Real-Time Systems". Ph.D. Thesis, Eindhoven University. 1991.

[MJ90] A. Moitra, M. Joseph, "Implementing Real-Time Systems by Transformation", in "Real-Time Systems, Theory and Practice", H.S.M. Zedan (ed). Elsevier Science Publishers B.V. 1990.

[Mor90] C. Morgan. "Programming from Specifications". Prentice-Hall. 1990.

[MRG88] C. Morgan, K. Robinson, P. Gardiner. "On the refinement calculus". Technical Report PRG-70. Programming Research Group. 1988

[Ost89] J. Ostroff. "Temporal Logic for Real-Time Systems". Research Studies Press. 1989.

[Sch92] D. J. Scholefield, "The Formal Refinement of Real-Time Systems". University of York Department of Computer Science Ph.D. Thesis. 1992. *to appear.*

[Schn90] S. Schneider, "Correctness and Communication in Real-Time Systems". Oxford University, Ph.D. Thesis PRG-84. 1990.

[SETL85] E. Schonberg, D. Shields, "From Prototype To Efficient Implementation: a Case Study Using SETL and C". Courant institute of mathematical sciences, Dept of computer science, New York University. 1985.

[SZ91] D. Scholefield, H.S.M. Zedan, "The Temporal Agent Model: Theory and Practice", University of York Technical Report YCS-163. 1991.

[SZ92] D. Scholefield, H.S.M. Zedan, "TAM: A Formal Framework for the Development of Distributed Real-Time Systems", in "Formal Techniques in Real-Time and Fault-Tolerant Systems", Nijmegen LNCS 571. 1992.

[TM89] C. Tofts, F. Moller. "A Temporal Calculus Of Communicating Systems". Edinburgh University, Technical Report LFCS-89-104. 1989.

[Tof90] C. Tofts. "Timed Concurrent Processes". BCS-Leicester Workshop in Semantics for Concurrency. Springer-Verlag 1990.

A Refinement

We define the refinement relation by:

$\mathcal{A} \sqsubseteq \mathcal{B}$ iff for any interval $[\delta, u]$, post-condition ϕ, communication predicate ψ, then:

$$wp^u_\delta(\mathcal{A}, \phi, \psi) : w.f \ \Rrightarrow \ wp^u_\delta(\mathcal{B}, \phi, \psi) : w.f \quad (i)$$
$$wp^u_\delta(\mathcal{B}, \phi, \psi) : w.c \ \Rrightarrow \ wp^u_\delta(\mathcal{A}, \phi, \psi) : w.c \quad (ii)$$

We define a subset of the refinement calculus where the numbering of laws is taken from [Sch92].

We denote bi-refinement, i.e. $\mathcal{A} \sqsubseteq \mathcal{B} \ \wedge \ \mathcal{B} \sqsubseteq \mathcal{A}$ as $\mathcal{A} = \mathcal{B}$.

Law 1. (Assignment)

$$1.2 \ \{x\} : x@t' = e@t \ \wedge \ t' \geq t+1 \ = \ x := e$$

Law 2. (Input)

$$2.2 \ \{x\} : \bigvee_{\sigma \in [t+1, t']} s@\sigma = x@t' \ = \ x \leftarrow s$$

Law 3. (Output)

$$3.2 \ \{s\} : write(s, x@t, t, t') \ = \ x \rightarrow s$$

Law 4. (Sequence)

$$4.1 \ [n]abort : \mathcal{A} \ = \ abort$$

$$4.2 \ [n]\mathcal{A} : abort \ = \ abort$$

$$4.3 \ [0]skip : \mathcal{A} \ = \ \mathcal{A}$$

$$4.4 \ ;_I \ [n_i](x := e_i) : A \ = \ [\,{}^{\Sigma}_{i \in I} n_i](x := \tilde{e_i} \mid \delta \# I) : \mathcal{A}$$

if $\forall i \in I(n_i > 0)$

(where $\tilde{e_i}$ is the ordered composition of expressions e_i)

4.5 $w:\Phi \;\sqsubseteq\; [n](w:\Phi_0):(w:\Phi_1)$
(if $\exists m(m \in I \;\wedge\; \Phi_0[m/t'] \;\wedge\; \Phi_1[m/t]) \;\equiv\; \Phi$
if $I=[t,min(t+n,t')]$ and
$\Sigma(w:\Phi_0)+\Sigma(w:\Phi_1) \;=\; \Sigma(w:\Phi)$

Law 5. (Concurrency)
5.1 $\mathcal{A}|\mathcal{B} \;=\; \mathcal{B}|\mathcal{A}$
5.2 $\mathcal{A}|(\mathcal{B}|\mathcal{C}) \;=\; (\mathcal{A}|\mathcal{B})|\mathcal{C}$
5.3 $\mathcal{A}|skip \;=\; skip$
5.4 $\mathcal{A}|abort \;=\; abort$
5.5 $\bigcup_{i\in I} w_i : \bigwedge_{i\in I}\Phi_i \;=\; \prod_{i\in I} w_i:\Phi_i$
if w_i are disjoint and $w_i:\Phi_i$ are interference free

Law 6. (Timeout)
6.4 $w:\Phi \;\sqsubseteq\; w:\Phi_0 \triangleright^s_n w:\Phi_1$
if $\Phi \;\equiv\; t' \geq t+n \;\wedge\; ((stable(s,t,t+n) \wedge$
$all_stable(w,t,t+n) \;\wedge\; \Phi_0[t+n/t]) \vee$
$(\bigvee_{\sigma\in[t,t+n]}(s@\sigma.1=\sigma \;\wedge\; stable(s,t,\sigma\ominus 1)$
$\wedge\; all_stable(w,t,\sigma) \;\wedge\; \Phi_1[\sigma/t]))$

Law 7. (Guards)
7.2 $true \Rightarrow \mathcal{A} \;=\; [\Delta(true)]\delta\Delta(true):\mathcal{A}$
7.3 $false \Rightarrow \mathcal{A} \;=\; \delta\Delta(false)$
7.4 $b_1 \Rightarrow \mathcal{A} \;\sqcup\; b_2 \Rightarrow \mathcal{A} \;=\; (b_1 \vee b_2) \Rightarrow \mathcal{A}$
(if $\Delta(b_1)+\Delta(b_2)=\Delta(b_1 \vee b_2)$
7.5 $b_1 \Rightarrow (b_2 \Rightarrow \mathcal{A}) \;=\; (b_1 \wedge b_2) \Rightarrow \mathcal{A}$
(if $\Delta(b_1)+\Delta(b_2)=\Delta(b_1 \wedge b_2)$
7.6 $[n](\bigsqcup_{i\in I} g_i \Rightarrow \mathcal{A}_i):\mathcal{B} \;=$
$\bigsqcup_{i\in I} g_i \Rightarrow [n-\overset{\Sigma}{}_{i\in I}\Delta(g_i)]\mathcal{A}_i : B$
if $\overset{\Sigma}{}_{i\in I}\Delta(g_i) \leq n$
7.9 $w:\Phi \;\sqsubseteq\; \bigsqcup_{i\in I} g_i \Rightarrow w_i:\Phi_i$
if $\Phi \;\equiv\; \exists\sigma(\sigma = \overset{\Sigma}{}_{i\in I\Delta(g_i)} \wedge$
$((\bigvee_{i\in I}(g_i@\delta \;\wedge\; \Phi_i[t+\sigma/t]) \vee$
$(\bigwedge_{i\in I}\neg g_i@\delta \;\wedge\; all_stable(w,t,t+\sigma)))$
and for each i, $\Sigma(w_i:\Phi_i) \;=\; \Sigma(w:\Phi)-\overset{\Sigma}{}_{i\in I}\Delta(g_i)$

Law 8. (Recursion)
8.1 $\mu X_0.F(X) \;=\; skip$
8.2 $\mu X_1.F(X) \;=\; F[skip/X]$
8.3 $\mu X_{n+1}.F(X) \;=\; F(\mu X_n.F(X))$
8.4 $w:\Phi \;\sqsubseteq\; \mu X_n.[p](w:\Phi'):X$
if $\Phi \;\equiv\; \exists\tilde{m_i}((\bigwedge_{i\in[1,n]}\Phi'[m_i/t'][m_{i-1}/t])[t/m_n]\wedge$
$\forall i\in[1,n](m_i \leq m_{i-1}+p)\wedge$
$t' \geq m_n \;\wedge\; all_stable(w,m_n,t'))$

Law 9. (Variable Restriction)
9.4 $w:\exists x(\Phi) \;=\; (x)(w\cup\{x\}:\Phi)$ (if $x \notin w$)

Law 10. (Shunt Restriction)
10.3 $w:\exists s(\Phi) \;=\; (w:\Phi)/s$

Law 11. (General)
11.1 $skip \;=\; \delta 0$
11.2 $\emptyset : t' \geq t+n \;=\; \delta n$

Law 12. (Specification)
12.1 $w:\Phi \;=\; w:\Phi[x@t/x@t']$ (if $x \notin w$)
12.2 $w:\Phi \;=\; w:\Phi[x@t'/x@t]$ (if $x \notin w$)
12.3 $w\cup\{s\}:\Phi \;\wedge\; stable(s,t,t') \;=\; w-s:\Phi$
(if s not in Φ)
12.4 $w:\Phi \;=\; w:\Phi \;\wedge\; t' \geq t+n$ (if $\Sigma(w:\Phi)=n$)
12.5 $w:\Phi_0\vee\Phi_1 \;\sqsubseteq\; w:\Phi_0$
(if $\Sigma(w:\Phi_0)=\Sigma(w:\Phi_1\vee\Phi_0)$)
12.6 $w:\Phi \;\sqsubseteq\; w:\Phi'$
(if $\Phi' \not\equiv \Phi$ and $\Sigma(w:\Phi)=\Sigma(w:\Phi')$)
12.7 $w\cup\{x\}:\Phi \;\wedge\; stable(x,t,t') \;=\; w:\Phi$ (if $x \notin \Phi$)