A semantics of core AUTOSAR

Johan Nordlander Patrik Jansson

Department of Computer Science and Engineering, Chalmers University of Technology

***Abstract*—The AUTOSAR system architecture for the automotive industry is rich in features and large in size, but lacks a comprehensive definition that allows its Software Components and Runtime Environment to be treated as an abstract programming model in its own right. As a result, AUTOSAR components are hard to analyze, test and evaluate without first committing to a particular execution platform and component context. This paper addresses this shortcoming by means of a formalization of the core AUTOSAR specification as a labelled transition system. The resulting semantic rules can be used to clarify the specification, emphasize design tradeoffs and serve as a foundation for tools that fulfill the AUTOSAR promise of platform- and vendor-independent automotive software development.**

I. Introduction

The AUTOSAR standard is an open software component architecture for the automotive industry [1]. Its main purpose is to enable interoperability of software modules among different vendors and on heterogeneous platforms, via an extensive set of standardized interfaces and libraries, and a common software development methodology.

The standard has a rather wide scope and covers many features normally associated with complex operating systems, like I/O abstraction, concurrency, communication, distribution and real-time predictability. Unlike existing operating systems, however, AUTOSAR is not de facto defined in terms of a particular implementation. Instead, an explicit goal of AUTOSAR is to constitute an abstract specification that allows multiple and competing realizations, and even systems built as an assembly of fragments from many different (and competing) vendors. Such a goal naturally puts a heavy focus on the standard specification itself.

Unfortunately, the AUTOSAR specification is not very rigorous, despite a sheer size of more than 12,500 pages of text and UML diagrams in total. It is also not very abstract, in that it makes frequent references to assumed implementation techniques for the purpose of defining its semantics. In practice, the AUTOSAR standard becomes blurred with the specific behavior of one's chosen platform and development tools. And because the standard is open to interpretation, the interoperability of software components across tools and platforms is often seriously hampered. What is more, a single software component cannot easily be studied and understood in isolation, since its interactive behavior is only indirectly defined in terms of the concrete C-code and OS tasks that realize it and its interaction environment.

In this paper we take a first step towards a remedy to these problems, by contributing a formal specification of a substantial core of the AUTOSAR standard. AUTOSAR itself is summarized in Section II. The formalization covers most of the Software Component Template and its accompanying Run-Time Environment (RTE) (Section III), and is able to directly express every legal way a system of software components can evolve at run-time on an arbitrarily fast platform (Section IV). It can thus serve as a basis for both a concrete AUTOSAR implementation on a specific platform (whose supported behaviors must be a subset of those defined by the formal semantics), and a platform-independent AUTOSAR simulator (where behaviors from the legal set can be picked at random). Non-determinism and concurrency is emphasized in Section V, together with an example of an unfortunate consequence of the specified standard. Section VI identifies some areas where the AUTOSAR documents has been found unclear; here our chosen readings are put in relation to alternative interpretations. Section VII, finally, gives a detailed account on the scope limitations we have imposed on our work.

The semantics has been formalized with the intent to accurately capture the informal meaning of the AUTOSAR standard, although mistakes and misunderstandings are certainly both possible and plausible. A formal notation is nevertheless a good starting point for any discussion on the resolution of such issues. The semantics is furthermore written to be unambiguous, except where concurrent execution should allow for more than one observed behavior. Examples of questions that can be firmly resolved on basis of our formalized semantics include:

- *How many runnable instances will a sequence of received values give rise to?*
- *What values may ever be returned when an inter-runnable variable is first read?*
- *At what times will two timing events coincide, if at all?*

II. AUTOSAR

An AUTOSAR model is primarily a structure of interconnected **software components** (SWCs) [2]. Links between SWCs are called **connectors**, which attach to **ports** exposed by the SWCs. A port is either **required** or **provided**, and then classified according to what **interface** it exposes: **sender-receiver**, **client-server**, **trigger** and **mode-switch** are common interface types. A sender-receiver interface is an aggregation of one or more **data-elements**, with each element characterized by the type of data it carries and optional further communication specification details. A client-server interface likewise consists of one or more **operations**, that each gives the signature of a procedure call and the possible errors that may result.

Some components merely act as containers of other components structures, these are called **composition** SWCs. In contrast, **atomic** SWCs define their own behavior, primarily in terms of **runnables** and **inter-runnable variables**. A runnable denotes a sequential and (normally) terminating piece of code, although its actual behavioral definition is typically relegated to external C or Matlab files rather than being part of the AUTOSAR model proper. Instead, an AUTOSAR runnable only provides constraints on what ports and inter-runnable variables the actual implementation can access; as well as **event** declarations that determine under what conditions it will get executed.

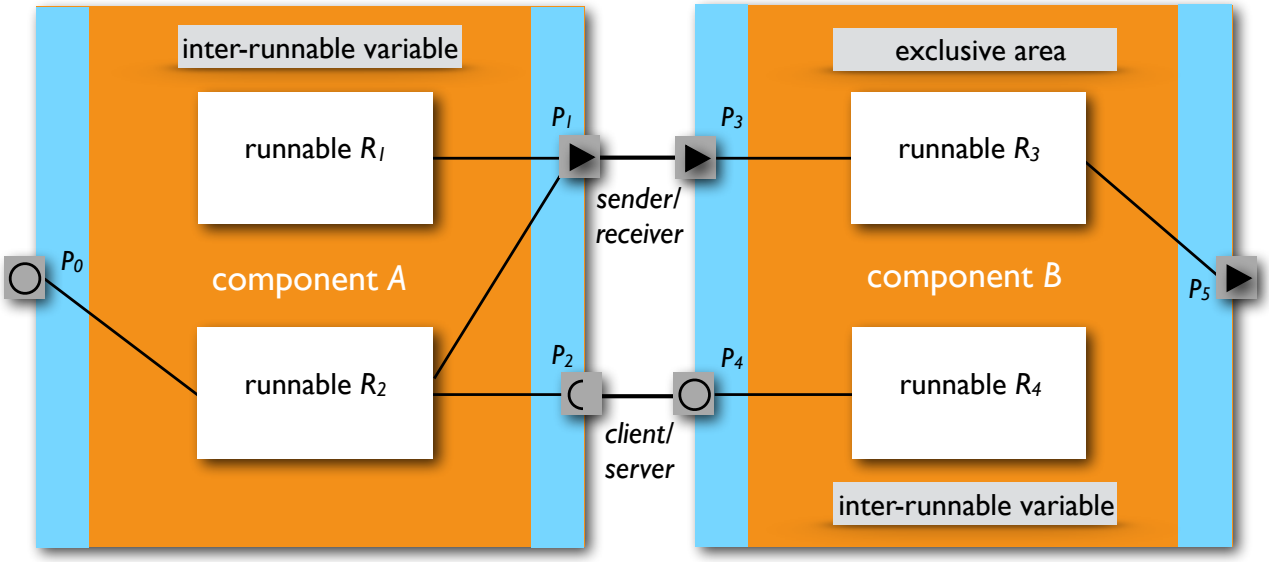


Fig. 1. Example of a graphical AUTOSAR software components model.

A runnable under execution is called a **runnable instance**, and several instances may execute concurrently even if they belong to the same SWC. Each runnable indicates whether or not it may be instantiated concurrently with itself. Runnable code can also use **exclusive areas** (a form of mutex semaphores) to further control concurrent behavior.

To interact with ports, inter-runnable variables and exclusive areas, runnables use what is abstractly known as the **virtual function bus** (VFB). Concretely, the VFB takes the shape of a C programming interface to the **run-time environment** (RTE), custom-generated for each runnable on the basis of its access constraints [3]. Behind the RTE, a collection of OS kernels, communication stacks and other basic software modules implement the VFB services for the platform at hand. Complete AUTOSAR designs also typically contain **ECU** and **task assignments**, which are manually built tables that control how SWCs and runnables map onto the available hardware and OS resources, respectively.

AUTOSAR models are completely static, which means that all components, runnables, ports, connectors, etc, are created ahead of execution time. Runnable instances are an exception, but they are only identified by AUTOSAR for conceptual purposes and are never part of any model syntax. This static nature renders AUTOSAR models very suitable to graphical notations, which by far is the most widespread representation format for AUTOSAR models. A graphical model example is shown in Figure 1.

III. Syntax and Preliminaries

The formal approach we have chosen is that of a typical process calculus [9], where a system of concurrent processes evolves in a sequence of atomic steps, as determined by a global set of transition rules. Our process terms correspond to the individual state-carrying parts of an AUTOSAR system, like the runnables, runnable instances, inter-runnable variables and port elements. Each such term is also assigned a constant address parameter for identification purposes.

Components do not carry any dynamic state besides the state of their constituent parts, so they need no explicit representation as process terms. However, in order to still be able to talk about model elements using their locally scoped names, we employ a hierarchical addressing scheme, such that address $i.r$ means runnable r of component i and $i.e$ means port element e within component i. We use the following meta-variable conventions:

$$
\begin{array}{rcl}
i & \in & \text{Component names} \\
r & \in & \text{Runnable names} \\
e & \in & \text{Sender-receiver port element names} \\
o & \in & \text{Client-server port operation names} \\
s & \in & \text{Inter-runnable variable names} \\
x & \in & \text{Exclusive area names}
\end{array}
$$

$$
\begin{array}{rcl}
a, b & \in & \text{Addresses} \\
a, b & ::= & i.r \mid i.e \mid i.o \mid i.s \mid i.x
\end{array}
$$

AUTOSAR elements and operations are actually only unique within the *ports* they belong to, so technically we ought to address them by reference to an intended *port name* as well, as in $i.p.e$ or $i.p.o$ for some port p. However, port names play no role of their own in our study, so we choose to leave them implied by the element and operation names we reference. Moreover, component names i can also be considered to have a nested internal structure (component i_1 within component i_2 within component i_3, etc). We can ignore this detail too, though, as the hierarchical layout of components in an AUTOSAR system has no run-time implications [2, ch. 3.3.1].

The process terms of our calculus are as follows:

- **runnable**$(i.r, t, act, n)$
  Dynamic state for runnable r of component i, indicating n current instances, t seconds left until next possible instantiation, and activation state act.
- **rinst**$(i.r, c, xs, code)$
  An instance of runnable r of component i, currently exe-

cuting $code$ and owning the exclusive areas xs, possibly running on behalf of client c.
- **irv**$(i.s, v)$
  Inter-runnable variable s of component i, currently holding value v.
- **excl**$(i.x, u)$
  Exclusive area x of component i, with boolean occupation state u.
- **delem**$(i.e, u, v)$
  Non-queued sender-receiver port element e of component i, currently holding value v with boolean update status flag u.
- **qelem**$(i.e, n, vs)$
  Queued sender-receiver port element e of component i with maximum capacity n, currently holding value sequence vs.
- **oper**$(i.o, m, srv)$
  Client-server port operation o of component i, currently holding sequence number m and call state srv.
- **timer**$(i.r, t_p, t)$
  Periodic timer for runnable r of component i, with a period of t_p and t seconds left of its current cycle.

In the untyped setting of this report, a value v can be any data item computed and communicated by an AUTOSAR system, including the unit value **void** and the error codes that might be returned from RTE calls [3, ch. 5.5.1]. Meta-variables n and m stand for natural numbers, while u and t range over boolean values and (floating point) time values, respectively. The client c of a runnable instance is a tuple holding the address, sequence number and argument value of the current invocation (if the runnable was invoked by a client-server call), or is **void** otherwise. An activation state act either toggles between **idle** and **pending**, or is a sequence of client tuples (in the case of a runnable triggered by client-server calls). A call state srv alternates between indicating an ongoing call with a timeout and a finished call with a result value. Formally,

$$\begin{array}{rcl} v & \in & \text{Values} \\ n, m & \in & \text{Natural numbers} \\ u & \in & \text{Boolean values} \\ t & \in & \text{Time values} \\ c & \in & \text{Clients} \\ act & \in & \text{Activation states} \\ srv & \in & \text{Call states} \end{array}$$

$$\begin{array}{rcl} c & ::= & (i.o, m, v) \mid \textbf{void} \\ act & ::= & \textbf{idle} \mid \textbf{pending} \mid cs \\ srv & ::= & \textbf{calling}(t) \mid \textbf{done}(v) \end{array}$$

We write vs for a sequence of values v, cs for a sequence of clients c, etc. Sequences may be empty, which is written ϵ. By $|vs|$ we mean the length of sequence vs. Left (right) concatenation of an element with a sequence is written $v{:}vs$ ($vs{:}v$). We use meta-variable conventions to resolve whether concatenation implies a fixed number of elements (as in $v_1{:}v_2$) or is open-ended (as in $v{:}vs$ or $vs{:}v$).

The $code$ part of a runnable instance should technically be a representation of the C or Matlab implementation that must accompany an AUTOSAR runnable definition once it is complete. However, since our task in this report is not to investigate the semantics of these languages in any detail, a more abstract notion of executable code will be required.

To this end, we have chosen to ignore all internal computations and just capture the observable effects of runnable execution as a pure string of RTE (VFB) calls [3, ch. 5.6]. And because the result from an RTE call may in general affect subsequent behavior, we use a continuation-passing style [11] where every $code$ term except the final return command literally contains the commands that may follow, as a continuation parameter k.

$$\begin{array}{rcl} code & \in & \text{Code} \\ k & \in & \text{Values} \to \text{Code} \end{array}$$

$$\begin{array}{rcl} code & ::= & \text{enter}(x, k) \\ & \mid & \text{exit}(x, k) \\ & \mid & \text{irvWrite}(s, v, k) \\ & \mid & \text{irvRead}(s, k) \\ & \mid & \text{send}(e, v, k) \\ & \mid & \text{receive}(e, k) \\ & \mid & \text{write}(e, v, k) \\ & \mid & \text{read}(e, k) \\ & \mid & \text{invalidate}(e, k) \\ & \mid & \text{isUpdated}(e, k) \\ & \mid & \text{call}(o, v, k) \\ & \mid & \text{result}(o, k) \\ & \mid & \text{return}(v) \end{array}$$

In contrast to code sequences, process terms are completely unordered. This is expressed in the standard process calculus style using an associative operator $\mid$ that allows arbitrary sets of primitive processes to be composed in parallel. This operator also has a left and right unit in the form of process **0**, which stands for the empty, or terminated, process. The complete grammar for our process terms thus looks as follows:

$$\begin{array}{rcl} p, q & ::= & \textbf{runnable}(a, t, act, n) \\ & \mid & \textbf{rinst}(a, c, xs, code) \\ & \mid & \textbf{irv}(a, v) \\ & \mid & \textbf{excl}(a, u) \\ & \mid & \textbf{delem}(a, u, v) \\ & \mid & \textbf{qelem}(a, n, vs) \\ & \mid & \textbf{oper}(a, m, srv) \\ & \mid & \textbf{timer}(a, t_p, t) \\ & \mid & p \mid q \\ & \mid & \textbf{0} \end{array}$$

IV. Semantics

Our semantic approach defines the meaning of an AUTOSAR system as the possible chains of state changes that can be applied to the system's initial state; or equivalently, as the set of *traces* that can be generated from the initial process term p_0. A trace is a possibly infinite sequence of transitions

$$p_0 \xrightarrow{l_1} p_1 \xrightarrow{l_2} p_2 \xrightarrow{l_3} \cdots$$

where each step

$$p_{i-1} \xrightarrow{l_i} p_i$$

states that the system state captured as process term p_{i-1} can evolve into state p_i by means of a single transition labelled l_i.

The label l of a transition can essentially be of two kinds, indicating that the affected process either "says" or "hears" something during the transition. Such a label always consists of an address a paired with some additional detail d. There is also a special form of "hearing" that denotes letting time pass

and which does not carry any address. The different labels are captured in the following grammar.

l	$::=$	$a!d$	Say a and d
	$\mid$	$a?d$	Hear a and d
	$\mid$	$\delta(t)$	Let t seconds pass

The possible forms of the extra label detail d will be introduced as the different transition rules are defined.

Two processes p and q can make transitions in parallel if they agree on what is being said or heard. Just as the saying/hearing intuition suggests, at most one of the processes can have the saying role in such an agreement. The following set of inference rules capture this intuition formally.

$$p_1 \mid q_1 \xrightarrow{a!d} p_2 \mid q_2 \quad \text{if } p_1 \xrightarrow{a!d} p_2 \wedge q_1 \xrightarrow{a?d} q_2 \quad (1)$$

$$p_1 \mid q_1 \xrightarrow{a!d} p_2 \mid q_2 \quad \text{if } p_1 \xrightarrow{a?d} p_2 \wedge q_1 \xrightarrow{a!d} q_2 \quad (2)$$

$$p_1 \mid q_1 \xrightarrow{a?d} p_2 \mid q_2 \quad \text{if } p_1 \xrightarrow{a?d} p_2 \wedge q_1 \xrightarrow{a?d} q_2 \quad (3)$$

$$p_1 \mid q_1 \xrightarrow{\delta(t)} p_2 \mid q_2 \quad \text{if } p_1 \xrightarrow{\delta(t)} p_2 \wedge q_1 \xrightarrow{\delta(t)} q_2 \quad (4)$$

The effect of these agreement rules closely resembles the idea of a *broadcast*: what one process says may be heard by every other process in a large parallel composition [10]. This behavior serves us well, because in general, what one AUTOSAR runnable does may have impact on many (if not all) parts of the executing system. It is, however, important not to confuse this broadcasting notion with any particular form of AUTOSAR communication [2, ch. 4.2]. The way transition labels distribute over the parallel composition operator is merely a technical aspect of our process calculus, and will be used to express several different AUTOSAR communication semantics, among other things.

The initial process p_0 is determined fully by the AUTOSAR model under study, as the parallel composition of the following terms:

1) For each runnable r of each component i, a term

   $$\textbf{runnable}(i.r, 0, act, 0)$$

   where act is ϵ if r is triggered by an *OperationInvokedEvent*; else act is **pending** if r is triggered by an *InitEvent*; otherwise act is **idle**.
2) For each inter-runnable variable s of each atomic component i, a term

   $$\textbf{irv}(i.s, v)$$

   where v is the *initValue* attributed to s if it exists, otherwise v is **void**.
3) For each exclusive area x of each atomic component i, a term

   $$\textbf{excl}(i.x, \textbf{false})$$

4) For each *required* port element e of each atomic component i, a term

   $$\textbf{qelem}(i.e, n, \epsilon)$$

   if the *swImplPolicy* attribute of element e is *queued* (with capacity n); otherwise, a term

   $$\textbf{delem}(i.e, \textbf{false}, \textbf{neverReceived})$$

5) For each *required* port operation o of each atomic component i, a term

   $$\textbf{oper}(i.o, 0, \textbf{done}(\textbf{void}))$$

6) For each *TimingEvent* of each runnable r of each atomic component i, a term

   $$\textbf{timer}(i.r, t_p, 0)$$

   where t_p is the period of the event.

The AUTOSAR naming scheme [2, ch. 3.2.5] guarantees that all processes terms in such an initial composition will carry unique addresses, with the exception of timer terms which share addresses with the runnables they belong to. We will later see how runnable instances reuse the address of their runnables in the same manner.

Some further static model information will also be referenced by the transition rules. We refer to the connectors of a model using a binary relation $\Rightarrow$ on port elements and operations, where the arrow points in the *provided-to-required* direction [2, ch. 3.3.3]. Although connectors are technically defined for ports, we assume $\Rightarrow$ is lifted to the common elements and operations of connected ports, and that it is transitively closed (i.e., expresses end-to-end connectivity across *Delegation*- as well as *AssemblySwConnector*s).

We expect the implementation of a runnable to be available as a continuation function k, even though the real AUTOSAR model will be referring to concrete functions in external C files [3, ch. 5.1.1]. We will further abstract away from actual numbers of input and output parameters to these functions by assuming that records (structs) are used as the single continuation argument and result whenever the arity so requires. Absent arguments or results will be replaced by the unit value **void**.

We write *implementation*$(i.r, k)$ to state the assumption that the static AUTOSAR model under study assigns implementation function k to runnable $i.r$. Similar notations will be used to express other references to the underlying static model; for example *DataReceivedEvent*$(i.r, i.e)$ to state that the model allows runnable $i.r$ to be triggered by data reception events on port element $i.e$. Although the chosen notation is sometimes significantly shorter than the UML/XML-based syntax used in the AUTOSAR specification, the intended meaning should nevertheless be clear.

The semantic transition rules and axioms are written in full in Appendix A. We will discuss each definition group in turn, starting with the relatively simple behavior of inter-runnable variables and exclusive areas. In most cases, the driving force behind a transition is some runnable instance wishing to say something which other processes either react to or ignore. How the runnable instances themselves come into existence will be detailed in Section IV-G.

A. Inter-runnable variables

An inter-runnable variable (irv) represents state that is local to a particular component and accessible by all runnables of that component [3, ch. 4.3.3.1]. Axioms (5) and (7) show the *say* transitions taken by a runnable instance wishing to read and write an irv s. In both cases the embedded continuation k is applied to the command result. The corresponding *hear* transitions of an irv appear as axioms (6) and (8). Thanks to inference rule (1), a runnable instance and an irv can now

engage in a parallel transition, for example expressing the writing of shared data.

$$\textbf{rinst}(i.r, c, xs, \text{irvWrite}(s, v, k)) \mid \textbf{irv}(i.s, \_) \xrightarrow{i.s!\text{IRVW}(v)} \textbf{rinst}(i.r, c, xs, k(\textbf{void})) \mid \textbf{irv}(i.s, v)$$

The corresponding read transition looks as follows:

$$\textbf{rinst}(i.r, c, xs, \text{irvRead}(s, k)) \mid \textbf{irv}(i.s, v) \xrightarrow{i.s!\text{IRVR}(v)} \textbf{rinst}(i.r, c, xs, k(v)) \mid \textbf{irv}(i.s, v)$$

Section IV-I will show how examples like these can be extended to the case when both participating processes are embedded in arbitrarily large parallel process compositions.

Note that in the last example above, the read value v flows from the irv process to the runnable instance, even though axiom (6) states that an irv actually *hears* the transition payload $\text{IRVR}(v)$. This apparent paradox is simply a reminder that the saying/hearing distinction of our calculus is entirely separate from the dataflow patterns it defines.

B. Exclusive areas

An exclusive area is the AUTOSAR equivalent of a binary semaphore, that can be acquired and released in an atomic fashion by competing runnable instances [3, ch. 4.2.2.5]. Axiom (9) states that a runnable instance wishing to enter exclusive area x may successfully proceed to its continuation k by broadcasting $i.x!\text{ENT}$. Axiom (10) expresses that exclusive area $i.x$ accepts hearing an ENT payload if it is currently not taken; i.e., carries the boolean state **false**. This makes the following parallel transition possible:

$$\textbf{rinst}(i.r, c, xs, \text{enter}(x, k)) \mid \textbf{excl}(i.x, \textbf{false}) \xrightarrow{i.x!\text{ENT}} \textbf{rinst}(i.r, c, x{:}xs, k(\textbf{void})) \mid \textbf{excl}(i.x, \textbf{true})$$

Exiting from an exclusive area is simply the reverse, as defined in axioms (11) and (12):

$$\textbf{rinst}(i.r, c, x{:}xs, \text{exit}(x, k)) \mid \textbf{excl}(i.x, \textbf{true}) \xrightarrow{i.x!\text{EX}} \textbf{rinst}(i.r, c, xs, k(\textbf{void})) \mid \textbf{excl}(i.x, \textbf{false})$$

Because axiom (10) is only defined for exclusive areas in the not taken state, there is no way to derive any parallel transition of the following form:

$$p \mid \textbf{excl}(i.x, \textbf{true}) \xrightarrow{i.x!\text{ENT}} \ldots$$

This means that any runnable instance in p wishing to enter $i.x$ is effectively blocked from making progress until some other process in p chooses to exit $i.x$. In the same manner, a process wishing to exit an exclusive area not taken is also effectively blocked. The conditions on exiting are actually stronger than those that govern entering, since axiom (11) is only applicable to a runnable instance for which the exited area x is at the top of its stack of acquired exclusive areas (as expressed by the sequence pattern $x{:}xs$). An exclusive area can thus only be exited by the process that entered it, and only in the reverse order of entering [3, ch. 5.6.28]. Section VI-E will bring up some alternatives to this semantics, and also discuss the interpretation of blocked processes in more detail.

C. Unbuffered sending/receiving

Axioms (13) and (15) define unbuffered inter-process communication from a runnable instance point of view. These transitions are just syntactic variants of shared variable reading and writing (axioms (5) and (7)). A data element (**delem**) process term contains two pieces of state in addition to its address: a communicated data value and a boolean flag for keeping track of unread writes [3, ch. 4.3.1.10.1]. Axioms (14) and (16) capture reads and writes to the data value and also set the update flag accordingly. The side-condition in axiom (16) makes the rule apply only if there is a static connection from the data element being written and the address of the matched term (recall that data element storage is associated with the receiving side of a connection). Because AUTOSAR allows sender-receiver communication patterns to be one-to-many (such as $i.e \Rightarrow a$ and $i.e \Rightarrow b$), parallel data element updates like the following are possible [3, ch. 4.3.1.4]:

$$\textbf{rinst}(i.r, c, xs, \text{write}(p.e, v, k)) \mid \textbf{delem}(a, \_, \_) \mid \textbf{delem}(b, \_, \_) \xrightarrow{i.e!\text{WR}(v)} \textbf{rinst}(i.r, c, xs, k(\textbf{void})) \mid \textbf{delem}(a, \textbf{true}, v) \mid \textbf{delem}(b, \textbf{true}, v)$$

As an aside, note that the ordering of terms in a parallel composition is made entirely irrelevant by the symmetric formulation of transition rules (1) and (2).

Axiom (17) provides another means of interpreting a data element write. It expresses that a runnable set up to trigger on data reception on required data element a should be marked as **pending** whenever a write occurs on an element connected to a. An example may be as follows (assuming $i.e \Rightarrow a$ and *DataReceivedEvent*(b, a)):

$$\textbf{rinst}(i.r, c, xs, \text{write}(p.e, v, k)) \mid \textbf{delem}(a, \_, \_) \mid \textbf{runnable}(b, t, \_, n) \xrightarrow{i.e!\text{WR}(v)} \textbf{rinst}(i.r, c, xs, k(\textbf{void})) \mid \textbf{delem}(a, \textbf{true}, v) \mid \textbf{runnable}(b, t, \textbf{pending}, n)$$

Transitions that actually create an instance of a pending runnable will be introduced in Section IV-G.

A data element can also be queried for its update status flag [3, ch. 5.6.35], and be marked as carrying no valid value [3, ch. 5.6.7]. Axioms (18)-(21) define the corresponding transitions.

D. Buffered sending/receiving

Buffered inter-process communication (axioms (22) - (30)) is just a minor variant of the unbuffered mechanism, where the single-value store of a **delem** term is replaced by a **qelem** process holding a sequence of values [3, ch. 4.3.1.10.2]. The buffer is consumed from the left (axiom (23)) and extended to the right (axiom (27)). When the buffer is empty, the special value (error code) **nodata** is returned to receiving runnable instances (axiom (24)). When the buffer is at its maximum capacity, the communicated value is simply dropped (axiom (28)).

However, the AUTOSAR specification requires the send command to inform its caller whether all connected buffers were able to successfully store the communicated value or whether some buffers had it dropped [3, ch. 5.6.5]. Such non-local

information can be expressed as an additional parameter as to the SND transition label, listing the addresses of connected **qelem** processes that have no spare capacity. The preconditions of axioms (27) and (28) ensure that as reflects the truth, and axioms (25) and (26) feed different error codes to the sender's continuation depending on whether as is empty or not. By requiring connectivity for all a in as in axiom (26), the possibility of letting irrelevant addresses in as cause bogus **limit** results is ruled out.

The following example illustrates a scenario where a sent value is only stored in a subset of the connected buffers. Assuming $i.e \Rightarrow a$ and $i.e \Rightarrow b$ (and let as be the singleton sequence b):

$$\begin{gathered}\mathbf{rinst}(i.r, c, xs, \text{send}(p.e, v, k)) \mid \\ \mathbf{qelem}(a, 2, \epsilon) \mid \mathbf{qelem}(b, 2, v_1{:}v_2) \\ \xrightarrow{i.e!\textsc{snd}(v,b)} \\ \mathbf{rinst}(i.r, c, xs, k(\mathbf{limit})) \mid \\ \mathbf{qelem}(a, 2, v) \mid \mathbf{qelem}(b, 2, v_1{:}v_2)\end{gathered}$$

E. Calling a server

The behavior of client-server communication in AUTOSAR [3, ch. 4.3.2] depends on whether a client lists the required server port operation as a synchronous or an asynchronous callpoint (which are mutually exclusive model attributes). In the former case, the call command is treated as if it were immediately followed by a command for retrieving the server result (axiom (31)), whereas in the latter case, the call succeeds immediately and the result has to be retrieved explicitly by the client's continuation code (axiom (32)). In both cases, however, an attempt to call the server before it is done processing a previous asynchronous call leads to immediate abortion with the error code **limit** (axiom (35)).

The AUTOSAR specification restricts client-server connections to the many-to-one pattern only, assigning a dedicated runnable responsible for implementing the offered service [2, ch. 4.3.2]. It furthermore forbids runnables to be triggered by any events other than operation invocations of compatible signatures, making it natural to associate the state needed to buffer up server invocations with the server runnables themselves. On the other hand, the specification also introduces timeouts [3, ch. 4.3.2.3] and sequence counters [3, ch. 4.3.2.6.1] that are private to each connection, which is why our calculus also needs process terms that correspond to each required (i.e., client-side) operation. Axiom (33) shows how the current sequence counter m blazes a call transition if the client-side operation is not already busy, while the busy case is detected in axiom (36). In axiom (34), a successful call transition causes the client identifier a to be buffered up in the server runnable together with sequence number m and call parameter v. Axiom (37) lets a server runnable ignore an aborted call attempt.

Here is an example of how a client runnable instance, a client port operation and a server runnable interact during a call transition (assuming *OperationInvokedEvent*(b, a), $a \Rightarrow i.o$, and *serverCallPointTimeout*$(i.o, t)$).

$$\begin{gathered}\mathbf{rinst}(i.r, c, xs, \text{call}(p.o, v, k)) \mid \\ \mathbf{oper}(i.o, m, \mathbf{done}(\mathbf{nodata})) \mid \\ \mathbf{runnable}(b, 0, cs, 0) \\ \xrightarrow{i.o!\textsc{call}(m,v)} \\ \mathbf{rinst}(i.r, c, xs, k(\mathbf{ok})) \mid \\ \mathbf{oper}(i.o, m, \mathbf{calling}(t)) \mid \\ \mathbf{runnable}(b, 0, cs{:}(i.o, m, v), 0)\end{gathered}$$

F. Passing back a server result

The *result* command can appear as an implicit effect of a previous call command, if the operation invoked is marked as an *synchronousServerCallPoint* (c.f. axiom (31)). It can also be referenced explicitly in a client runnable's code, if the current runnable lists the operation as an *asynchronousServerCallPoint* [3, ch. 5.6.14]. For the synchronous case, axiom (38) excludes **nodata** values, which effectively blocks progress of the caller until a server result distinct from **nodata** has been produced. The asynchronous case of axiom (39) has no value restrictions, so it will happily pass back the **nodata** tag as well as an indication that a result is not yet available.

The state keeping track of a call's status resides in the **oper** term of a connection. Axiom (40) defines that **nodata** is the result provided while a call is open, whereas axiom (41) returns the stored result once a call has completed.

A server runnable instance is identified by a client parameter distinct from **void**. Axiom (42) says that when such a runnable instance reaches the end of its code sequence, it must announce the produced value, together with the sequence number and **oper** address of its current invocation, before it is allowed to terminate (to be defined in Section IV-G). The addressed **oper** term reacts according to axiom (43), by completing the call, storing the produced value and increasing its sequence counter in preparation for the next call. However, should the **oper** timeout counter reach zero before a transition according to (43) can be taken, the client is obliged to unilaterally terminate the call and set the result to error code **timeout** (axiom (44)). Axiom (45) enables the completion of a call—successfully or via a timeout—to trigger a subscribing runnable (can be distinct from the original client).

Because of call timeouts, server runnables run the risk of producing results that the original client is not waiting for anymore. Detecting such mismatches is the job of the sequence numbers, as witnessed by the requirement in axiom (43): the sequence number returned by the server and the one expected by the client must be identical (variable m). However, there must also exist a means for server runnables to simply discard their results, otherwise they would not be able to terminate. Axiom (46) therefore defines an alternative transition for finished servers, applicable on the condition that the returned sequence number does *not* match what the client expects (axiom (47)). By means of axiom (48), subscribing runnables are allowed to ignore the corresponding transition.

An example of a normal server result interaction can look as follows:

$$\begin{gathered}\mathbf{rinst}(b, (i.o, 91, v_0), xs, \text{return}(v)) \mid \\ \mathbf{oper}(i.o, 91, \mathbf{calling}(t)) \\ \xrightarrow{i.o!\textsc{ret}(91,v)} \\ \mathbf{rinst}(b, \mathbf{void}, xs, \text{return}(\mathbf{void})) \mid \\ \mathbf{oper}(i.o, 92, \mathbf{done}(v))\end{gathered}$$

As a contrast, this is the behavior observed when a call timeout occurs:

$$\begin{gathered}\mathbf{rinst}(b, (i.o, 91, v_0), xs, \mathit{code}) \mid \\ \mathbf{oper}(i.o, 91, \mathbf{calling}(0)) \\ \xrightarrow{i.o!\textsc{ret}(91,\mathbf{timeout})} \\ \mathbf{rinst}(b, (i.o, 91, v_0), xs, \mathit{code}) \mid \\ \mathbf{oper}(i.o, 92, \mathbf{done}(\mathbf{timeout}))\end{gathered}$$

The processes are now in a state where runnable instance b is prohibited from doing a normal return. Termination of b is only possible via axioms (46) and (47).

$$\begin{gathered}\mathbf{rinst}(b, (i.o, 91, v_0), xs, \text{return}(v)) \mid \\ \mathbf{oper}(i.o, 92, \mathbf{done}(\mathbf{timeout}))\end{gathered}$$
$$\xrightarrow{i.o!\text{SKIP}(91)}$$
$$\begin{gathered}\mathbf{rinst}(b, \mathbf{void}, xs, \text{return}(\mathbf{void})) \mid \\ \mathbf{oper}(i.o, 92, \mathbf{done}(\mathbf{timeout}))\end{gathered}$$

G. Spawning and terminating

Runnables with an activation state distinct from **idle** or ϵ are amenable to instantiation [3, ch. 4.2.3]. Axiom (49) defines this crucial step for runnables triggered by all types of events except server invocation, which is instead handled by axiom (50). Both rules have many details in common:

- They apply only when the runnable instantiation timer (parameter two) has reached zero.
- They reset the timer to a model-defined *minimumStartInterval* to disable further instantiation immediately after a new instance is born.
- They are guarded by the condition that the current number of runnable instances (fourth parameter) is either zero, or the runnable has been defined to accept concurrent invocations.
- They make the runnable keep track of the new number of instances.
- They let the new instance inherit the address of its parent runnable.
- They create the new instance in parallel with its parent (runnable instances is actually the only example of dynamic process creation in this calculus).
- They initialize the new instance with code according to the model-defined implementation and an empty list of owned exclusive areas.

The differences all emanate from the server/non-server distinction:

- The non-server runnable instance gets a **void** client parameter whereas the server instance is assigned the client tuple from the head of the server's client queue.
- The non-server runnable toggles back to the **idle** activation state, while the server just chops of the head of its client queue.
- Only the server runnable instance is given a non-void code parameter, which is taken from the same client queue head.

The transition label of axioms (49) and (59) is noteworthy because it is not matched by a corresponding hearing transition; all other process terms just ignore the information that a new runnable instance has been spawned. An example of process instantiation within a parallel context will thus merely be a trivial variant of one of the instantiation axioms, and is therefore left out.

Complementing the notion of spawning is a mechanism for runnable instance termination. An instance with a **void** client parameter (either acquired natively, or via axioms (42) or (46)) transforms into the terminated process **0** as defined in axiom (51). Since this term is the unit of parallel process composition, it is silently absorbed by any other process in parallel with it. The instance termination is also noted by the associated runnable parent, which reacts by decrementing its current instance counter (axiom (52)). An example of this interaction follows below.

$$\mathbf{runnable}(b, t, act, 3) \mid \mathbf{rinst}(b, \mathbf{void}, \epsilon, \text{return}(v))$$
$$\xrightarrow{i.o!\text{TERM}}$$
$$\mathbf{runnable}(b, t, act, 2) \mid \mathbf{0}$$
$$\equiv$$
$$\mathbf{runnable}(b, t, act, 2)$$

H. Passing time

In common with most real-time process calculi, the concept of passing time in our calculus is kept separate from the computational work expressed by the say/hear transitions [7]. This has the advantage of freeing the semantics from possessing a particular computation speed, as any finite number of computation steps can be performed before time must advance. On the other hand, it also means that the limitations of a particular platform cannot be directly studied unless the calculus is complemented with some form of constraint on the work/time relationship. Such an extension is beyond the scope of the current report, though.

The passage of time is captured as a special transition relation $\xrightarrow{\delta(t)}$, whose only effect on the related process terms is to decrement any contained time variables by t. For some processes, time cannot advance until a particular work transition has been taken; this corresponds to time events that must be noted (although not necessarily acted upon) at the exact time instance when they occur. Yet other processes can only take time transitions, which simply means that they represent an idle system that currently has no work to do. Mostly, though, work and time transitions are simultaneously available, indicating that the semantics considers the slightly slower and slightly faster behavior alternatives to be equally correct.

Most process terms actually ignore the passage of time, which is expressed by axioms (53) to (58). In axiom (59), an **oper** term in the **calling** state lets time pass by decrementing its timeout counter. The counter is not allowed to go negative, though, which forces the **oper** term to handle the timeout once the counter reaches zero (c.f. axiom (44)). Axiom (60) defines a similar behavior for the start interval timer of a runnable, although this timer is also allowed to slip behind further time updates once it has reached zero (axiom (61)). The reason for this is that the interval timers do not enforce any particular action when they time out, they just enable new transition options that do not have to be taken immediately [3, ch. 4.2.3].

A timer process, finally, decrements its counter according to axiom (62). When the counter has become zero, the timer is forced to take the TICK transition of axiom (63), which also resets the counter to its static period value t_p. The net effect of such a transition is that the runnable to which the ticking timer belongs is put into the **pending** state (axiom (64)). An example of a transition sequence involving a time-activated

runnable follows below [3, ch. 4.2.2.8].

$$\begin{array}{c}
\mathbf{runnable}(a,0,\mathbf{idle},0) \mid \mathbf{timer}(a,2.5,0.3) \\
\xrightarrow{\delta(0.3)} \\
\mathbf{runnable}(a,0,\mathbf{idle},0) \mid \mathbf{timer}(a,2.5,0) \\
\xrightarrow{a!\text{TICK}} \\
\mathbf{runnable}(a,0,\mathbf{pending},0) \mid \mathbf{timer}(a,2.5,2.5) \\
\xrightarrow{a!\text{NEW}} \\
\mathbf{runnable}(a,0.8,\mathbf{idle},1) \mid \mathbf{timer}(a,2.5,2.5) \mid \\
\mathbf{rinst}(a,\mathbf{void},\epsilon,code_1) \\
\longrightarrow \cdots \longrightarrow \\
\mathbf{runnable}(a,0.8,\mathbf{idle},1) \mid \mathbf{timer}(a,2.5,2.5) \mid \\
\mathbf{rinst}(a,\mathbf{void},\epsilon,code_2) \\
\xrightarrow{\delta(0.8)} \\
\mathbf{runnable}(a,0,\mathbf{idle},1) \mid \mathbf{timer}(a,2.5,1.7) \mid \\
\mathbf{rinst}(a,\mathbf{void},\epsilon,code_2) \\
\longrightarrow \cdots \longrightarrow \\
\mathbf{runnable}(a,0,\mathbf{idle},1) \mid \mathbf{timer}(a,2.5,2.5) \mid \\
\mathbf{rinst}(a,\mathbf{void},\epsilon,\text{return}(\mathbf{void})) \\
\xrightarrow{a!\text{TERM}} \\
\mathbf{runnable}(a,0,\mathbf{idle},0) \mid \mathbf{timer}(a,2.5,1.7) \\
\xrightarrow{\delta(1.5)} \cdots \xrightarrow{\delta(0.2)} \\
\mathbf{runnable}(a,0,\mathbf{idle},0) \mid \mathbf{timer}(a,2.5,0) \\
\xrightarrow{a!\text{TICK}} \\
\cdots
\end{array}$$

I. Ignoring broadcasts

Although our chosen mechanism for coordinating transitions among process terms goes under the name *broadcast*, there is no implication that every process must react to everything being said — the intent is rather that a process term not directly or indirectly addressed in a transition should be allowed to ignore it. One could try to express this as some form of "catch-up" rule $p \xrightarrow{l} p$, that would apply only if none of the axioms (5) to (64) match p and l. However, such a rule would be too liberal, as it would also enable transitions that are deliberately omitted from the previous axioms, like $\mathbf{excl}(i.x,\mathbf{true}) \xrightarrow{i.x!\text{ENT}} \mathbf{excl}(i.x,\mathbf{true})$ (granting entry to an exclusive area already taken).

Instead we need a more restrictive rule, that applies to processes not addressed by earlier axioms but avoids those previously excluded by restrictive patterns and other side-conditions. To make this idea precise, however, we need to split the rule according to the different forms of p so that the addressing notion of each process type can be individually captured. Axioms (65) to (72) contain the resulting definitions.

Runnable instances and timers can ignore everything being said by others, as they define no hearing transitions proper that need to be excluded here (axioms (65) and (66)). Exclusive areas, inter-runnable variables and operation terms have hearing transitions previously defined, but only for label addresses matching their own; they can thus safely ignore transitions labelled with any other address. Axioms (70) to (72) constitute the non-trivial cases, because here there are multiple earlier definitions to exclude, which also link label and process addresses via static model references. Still, it can be verified that each of the axioms (70) to (72) is guarded by an exact negation of the addressing conditions of any previously defined hearing transition for the same process type.

As an concrete illustration, consider the example of Section IV-D. The axioms that enable each individual process transition in that example are (26), (27) and (28), respectively (and rules (1) and (3) allow them to be combined). In particular, axiom (28) is applicable because of the assumption $i.e \Rightarrow b$, which simultaneously blocks applicability of axiom (70). However, under the different assumption that there is no connection from $i.e$ to b, axioms (28) and (70) must trade places. This also forces the removal of b from the second argument of the SND payload (i.e., the list of connected buffers not able to store the value sent). As a consequence, axiom (26) is no longer applicable and must be replaced by axiom (25). The resulting parallel transition looks as follows, illustrating the correct generation of an **ok** result despite the presence of a full (but unconnected) buffer.

$$\begin{array}{c}
\mathbf{rinst}(i.r,c,xs,\text{send}(p.e,v,k)) \mid \\
\mathbf{qelem}(a,2,\epsilon) \mid \mathbf{qelem}(b,2,v_1{:}v_2) \\
\xrightarrow{i.e!\text{SND}(v,\epsilon)} \\
\mathbf{rinst}(i.r,c,xs,k(\mathbf{ok})) \mid \\
\mathbf{qelem}(a,2,v) \mid \mathbf{qelem}(b,2,v_1{:}v_2)
\end{array}$$

V. NON-DETERMINISM

The transition relation defined in Appendix A is non-deterministic; i.e., it may very well be that for some process p,

$$p \xrightarrow{l_1} p_1 \qquad \text{and} \qquad p \xrightarrow{l_2} p_2$$

are both possible. If it also holds that

$$p_1 \xrightarrow{l_2} q \qquad \text{and} \qquad p_2 \xrightarrow{l_1} q$$

are derivable, we note that process p may evolve into state q either via an l_1 step followed by l_2, or by first taking the l_2 step and then l_1. Such freedom of choice is commonly taken as the definition of *parallelism* in the process calculus literature — if the sequential ordering of steps l_1 and l_2 has no significance, they might just as well be considered to occur in parallel [9].

However, it may also be that two possible step orderings do not lead to identical results. That happens if, instead of the two transitions leading to q above, we have

$$p_1 \xrightarrow{l_2} q_1 \qquad \text{and} \qquad p_2 \xrightarrow{l_1} q_2$$

where q_1 and q_2 are "similar" but not identical states. This non-determinism also has a concrete interpretation, either reflecting the non-deterministic nature of a system's environment (messages l_1 and l_2 just happen to arrive in some particular order) or its internals (processor speed or a scheduling choice just happens to pick l_1 before l_2). It is one of the primary merits of the process calculus approach to allow the scope of such inherent non-determinism to be precisely defined, because although a given system state may render many different behaviors possible, many more are not. A formal semantics enables us to clearly draw the line between these groups of behaviors.

The AUTOSAR specification contains a very subtle sensitivity to this kind of non-determinism that is worth exemplifying. Consider a queued data element a together with a runnable b set up to trigger on data reception events for a. A scenario

where two values v_1 and v_2 are sent on a port element $i.e$ connected to a may lead to the following trace:

$$\begin{gathered}
\textbf{qelem}(a, 2, \epsilon) \mid \textbf{runnable}(b, 0, \textbf{idle}, 0) \\
\xrightarrow{i.e?\text{SND}(v_1,\epsilon)} \\
\textbf{qelem}(a, 2, v_1) \mid \textbf{runnable}(b, 0, \textbf{pending}, 0) \\
\xrightarrow{b!\text{NEW}} \\
\textbf{qelem}(a, 2, v_1) \mid \textbf{runnable}(b, 0, \textbf{idle}, 1) \mid \\
\textbf{rinst}(b, \textbf{void}, \epsilon, \mathit{code}) \\
\xrightarrow{i.e?\text{SND}(v_2,\epsilon)} \\
\textbf{qelem}(a, 2, v_1{:}v_2) \mid \textbf{runnable}(b, 0, \textbf{pending}, 1) \mid \\
\textbf{rinst}(b, \textbf{void}, \epsilon, \mathit{code}) \\
\xrightarrow{b!\text{NEW}} \\
\textbf{qelem}(a, 2, v_1{:}v_2) \mid \textbf{runnable}(b, 0, \textbf{idle}, 2) \mid \\
\textbf{rinst}(b, \textbf{void}, \epsilon, \mathit{code}) \mid \textbf{rinst}(b, \textbf{void}, \epsilon, \mathit{code})
\end{gathered}$$

That is, a resulting state where a holds the two received values together with two concurrent instances of b ready to start reacting upon the two reception events.

However, an alternative trace is equally valid for the same scenario:

$$\begin{gathered}
\textbf{qelem}(a, 2, \epsilon) \mid \textbf{runnable}(b, 0, \textbf{idle}, 0) \\
\xrightarrow{i.e?\text{SND}(v_1,\epsilon)} \\
\textbf{qelem}(a, 2, v_1) \mid \textbf{runnable}(b, 0, \textbf{pending}, 0) \\
\xrightarrow{i.e?\text{SND}(v_2,\epsilon)} \\
\textbf{qelem}(a, 2, v_1{:}v_2) \mid \textbf{runnable}(b, 0, \textbf{pending}, 0) \\
\xrightarrow{b!\text{NEW}} \\
\textbf{qelem}(a, 2, v_1{:}v_2) \mid \textbf{runnable}(b, 0, \textbf{idle}, 1) \mid \\
\textbf{rinst}(b, \textbf{void}, \epsilon, \mathit{code})
\end{gathered}$$

Here there is only one b instance created and none is pending, even though a correctly contains the two values received. The reason is that by swapping the order of the first instantiation and the second data reception steps, the system is put in a state where its inability to count pending instantiations [3, ch. 4.2.3] becomes fatal.

So, in an abbreviated form, both the following derivations are possible for the same reception event sequence:

$$\begin{gathered}
q \mid p \to \cdots \to q_1 \mid p_1 \mid p_{\text{inst}} \\
q \mid p \to \cdots \to q_1 \mid p_2 \mid p_{\text{inst}} \mid p_{\text{inst}}
\end{gathered}$$

That is, the system can respond by creating either one or two runnable instances, and both behaviors are equally correct according to the AUTOSAR semantics.

Unfortunately there is very little a programmer can do to influence which of these two traces is taken. Giving b a positive *minimumStartInterval* just guarantees that any subsequent events are lost during that interval. Long delays between reception events reduce the likelihood of losses, but the choice of when to take an instantiation step is still fully internal to the operating system scheduler. Moreover, when messages are produced by multiple parallel senders, maintaining control over their time distribution may be highly impractical. Non-concurrent runnables can work around the problem by iterating over all received data at each invocation, but that solution is not very attractive if concurrent processing of each received item is desired.

One could argue that replacing the **pending/idle** flag of runnables with a small counter variable would render the AUTOSAR event triggering mechanism much more robust. The standard explicitly excludes counting for all events but operation invocation, though, so the possibility of event losses is undeniably intentional [3, p. 128]. However, should this design decision ever be revisited, the two trace alternatives above could serve as a succinct problem characterization.

VI. Ambiguities and clarification proposals

The formal semantics presented in this report is aimed to be a faithful capture of the AUTOSAR specification, especially its Software Components template and RTE specification [2], [3]. Still, there are several areas where we have found the specification open to interpretation, or at least in need of some further clarification. We detail a few of these areas here, as an illustration to the kind of design decisions that may surface as the result of a formalization undertaking.

A. Data initialization

Unqueued data elements that are read before first written should return the error code **neverReceived** [3, ch. 5.6.10]. Since no exceptions to this rule are mentioned in the specification, our interpretation is that this holds even when a data element possesses an *initValue* attribute (initial **delem** state in Section IV). However, as this renders any such initial value assignments entirely redundant, an alternative interpretation would be to create a term

$$\textbf{delem}(i.e, \textbf{false}, v)$$

for each unqueued data element $i.e$, where v is the *initValue* of e if it exists, and **neverReceived** otherwise.

Inter-runnable variables can also be read before written, although here there is no special error code defined in the specification for the uninitialized case [3, ch. 5.6.26]. Section IV sets each **irv** contents to **void** to indicate a truly non-existing value, although an entirely reasonable alternative would be

$$\textbf{irv}(i.s, v)$$

where v is the *initValue* of s if it exists, and the error code **nodata** otherwise.

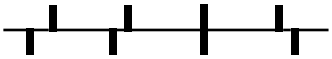
B. Activation of server runnables

The specification restricts client-server connections such that a client may not be connected to multiple servers, for the purpose of avoiding an operation call to be handled by more than one server runnable [3, ch. 4.2.3]. However, this restriction does not explicitly forbid multiple *OperationInvokedEvent*s referring to the same client-server operation to trigger separate runnables. Our formulation of client-server communication is based on the assumption that this is just an oversight, as allowing multiple events on a server operation would have the same problem with conflicting response values as would a one-to-many client-server connection pattern. It is actually not easy to see what could constitute an alternative interpretation here, without introducing a need for some new programming mechanism for selecting among server results.

C. Interleaving of timing events

Although the AUTOSAR specification only motivates timed runnable activation with the need to support plain periodic execution, there is really nothing in the specification that precludes such runnables from being triggered by additional events as well — including other *TimingEvent*s, perhaps even

with identical periods [2, table 7.1, ch. 7.2.3]. Our formulation naturally supports this generality, but it should be observed that the inability of AUTOSAR runnables to count pending activations (see Section V) also renders the resolution of coinciding timing events highly non-deterministic. As an example, consider the event patterns of two timers with distinct periods that occasionally coincide:



If the timers were activating two distinct runnables, the fragment above would be guaranteed to result in eight instantiations in total. But if directed towards a single runnable, the number of instances created could be seven as well as eight, with no means for the programmer to control the outcome.

This anomaly may lead to the conclusion that the AUTOSAR specification does not really intend to allow more than one timing event per runnable. Luckily, our formalization is also compatible with this interpretation, as the restriction only concerns what set of static models to expect as starting states, not their dynamic behavior.

Closely related to the interleaving of timing events, and of interest even if only distinct runnables are being triggered, is the notion of initial timer offsets. That is, at what time will each timer tick for the first time — immediately at startup or at some other (later) time? The AUTOSAR specification only introduces a concept of timing offsets in relation to the operating system tasks that are expected to implement runnables; the Software Component Template itself is totally silent on the issue [2]. Lacking an abstract definition of runnable offsets, we have chosen to simply fire all timers simultaneously at 0 seconds from startup (hence the value 0 in the initial **timer** term definition in Section IV). It is not hard to see that other values may be equally reasonable, though, like

$$\mathbf{timer}(i.r, t_p, t_p)$$

which lets each timer wait one full cycle before triggering for the first time. However, should we really want to take the assumed task offsets into account, the initial timer state would rather be

$$\mathbf{timer}(i.r, t_p, t_o)$$

for some non-negative t_o. One might even want to add the condition $t_o \leq t_p$, but it must be emphasized that all these initialization options are speculative — the AUTOSAR Software Component Template must be clarified before a sufficiently abstract model of timed behavior can be obtained.

D. Interpretation of data reception events

Activation of runnables due to *DataReceivedEvent*s is faithfully captured in the formal semantics for unqueued as well as queued data elements (c.f. axioms (17) and (29) with [2, ch. 7.5.1.6]). But for unqueued elements AUTOSAR also provides an invalidation operation, whose net effect is very similar to that of a data write (see axiom (21)). Whether data invalidation should count as a reception event is not clear from the specification; we have chosen not to include it as a conservative conjecture. Revising this choice would not be hard, though: all that is necessary is to add an axiom that mimics (17) but matches on INV rather than WR labels.

A similar specification ambiguity concerns *DataReceivedEvent*s in the presence of a full receiver buffer. We have chosen to block these events from activating connected runnables (axiom (30)), although the opposite choice could also be justified on grounds that activation of a receiver might be even more pressing when its buffer is overflowing. Expressing this formally would amount to simply deleting axiom (30) together with the last precondition of axiom (29).

E. Exclusive area nesting

The AUTOSAR specification is apparently ambiguous when it comes to exclusive areas, stating that "The RTE is not required to support nested invocations of [enter/exit] for the same exclusive area" [3, ch. 5.6.28-29]. Whether entering a previously acquired area should succeed or not is thus left entirely open, something we have taken as a justification not to support it formally (axioms (9) to (12)). Still, it is interesting to ponder what would be required to actually allow this optional behavior. One aspect is that the referenced exclusive areas themselves need not really be involved in the resulting transitions; they must already be in the occupied state they should retain. Another detail is that if a runnable is already holding its referenced exclusive area, both the enter and exit commands essentially turn into *no-operations*. Thus, the necessary additions to the axiom set would look something like this:

$$\mathbf{rinst}(i.r, c, xs, \text{enter}(x, k)) \xrightarrow{i.r!\text{NOP}} \mathbf{rinst}(i.r, c, x{:}xs, k(\mathbf{void})) \quad \text{if } x \in xs$$

$$\mathbf{rinst}(i.r, c, x{:}xs, \text{exit}(x, k)) \xrightarrow{i.r!\text{NOP}} \mathbf{rinst}(i.r, c, xs, k(\mathbf{void})) \quad \text{if } x \in xs$$

Another concern regarding exclusive areas is that although the specification clearly demands that exiting is done in the reverse order of entering, it does not provide any means for the exit command to report detected violations of that condition [3, ch. 5.6.29]. This means that if a runnable instance tries to exit exclusive area x, and x is not the latest exclusive area acquired by that instance, the operation is neither allowed to succeed, nor able to flag an error (due to a *void* return type). This leaves our transition semantics with no other option than to indefinitely block progress for such a runnable, which is the consequence of providing no transition alternative besides axiom (11) to runnable instances about to execute the exit command. Had the specification allowed for an error result, however, the following axioms could have been added to resolve the order violation cases:

$$\mathbf{rinst}(i.r, c, y{:}xs, \text{exit}(x, k)) \xrightarrow{i.r!\text{NOP}} \mathbf{rinst}(i.r, c, y{:}xs, k(\mathbf{orderErr})) \quad \text{if } x \neq y$$

$$\mathbf{rinst}(i.r, c, \epsilon, \text{exit}(x, k)) \xrightarrow{i.r!\text{NOP}} \mathbf{rinst}(i.r, c, \epsilon, k(\mathbf{orderErr}))$$

F. Failing runnables

A transition semantics like the one defined in this report provides rules for deriving the correct behaviors, while leaving the incorrect behaviors without derivation alternatives. This has the effect that a process term which attempts to do something illegal is assigned no meaningful remaining trace; it appears indefinitely stuck in its current state as if it were deadlocked, and can participate in subsequent transitions only by ignoring what is being said.

An alternative way of handling program failures in general is of course to throw exceptions or trigger some other non-local error-handling mechanism. AUTOSAR actually specifies a set of such generic error-handling approaches, but it does so in terms of implementation artifacts like tasks and memory partitions only, rather than the software component concepts under study in this report [5]. Still, the presence of exceptional errors raises a string of questions when combined with the concurrency and communication features of software components, and a clarification of this aspect could go a long way towards making AUTOSAR a self-contained programming model in its own right. We outline one possible way of formalizing the semantics of failing runnables below.

First the code syntax needs to be extended so that failing computations can be represented:

$$\begin{array}{rcl} code & ::= & \ldots \\ & | & \text{return}(v) \\ & | & \text{throw} \end{array}$$

A "throw" command has no continuation and is thus similar to the exception generating commands of many modern programming languages. It could be given an informative value parameter as well, but this will not be needed in our simplistic proposal. Throw commands can appear explicitly in the code to indicate software-detected errors, or they can be the result of special failure transitions, like the following alternative resolution of an exit order violation:

$$\begin{array}{l} \textbf{rinst}(i.r, c, y{:}xs, \text{exit}(x,k)) \xrightarrow{i.r!\text{FAIL}} \\ \textbf{rinst}(i.r, c, y{:}xs, \text{throw}) \qquad\qquad \text{if } x \neq y \end{array}$$

$$\begin{array}{l} \textbf{rinst}(i.r, c, \epsilon, \text{exit}(x,k)) \xrightarrow{i.r!\text{FAIL}} \\ \textbf{rinst}(i.r, c, \epsilon, \text{throw}) \end{array}$$

Because a failed runnable instance now has an explicit representation, we can specify details of how such a process should interact with its environment. We may for example wish to define that a special error code is returned if our runnable is a server running on behalf of a client, and that any held exclusive areas are automatically exited in the right order (c.f. axioms (42) and (11)):

$$\begin{array}{l} \textbf{rinst}(a, (b, m, \_), xs, \text{throw}) \xrightarrow{b!\text{RET}(m,\textbf{serverErr})} \\ \textbf{rinst}(a, \textbf{void}, xs, \text{throw}) \end{array}$$

$$\begin{array}{l} \textbf{rinst}(i.r, \textbf{void}, x{:}xs, \text{throw}) \xrightarrow{i.x!\text{EX}} \\ \textbf{rinst}(i.r, \textbf{void}, xs, \text{throw}) \end{array}$$

Finally, we can ensure that the instance counter of a runnable is properly decremented even if an instance terminates due to a failure (in interaction with axiom (52)):

$$\textbf{rinst}(a, \textbf{void}, \epsilon, \text{throw}) \xrightarrow{a!\text{TERM}} \mathbf{0}$$

Other design choices are of course perfectly possible, and more elaborate rules, for example involving an explicit exception-catching mechanism, could be considered as well. We must emphasize, though, that the rules in this section are just hypothetical examples; they are not based on the current AUTOSAR specification and are only presented to illustrate the succinctness and level of detail that can be obtained by a formal semantics approach.

VII. Limitations

We claim that the formalized AUTOSAR semantics of this report covers a substantial core of the Software Component Template and Specification of RTE [2], [3]. But these two documents alone add up to almost 2000 pages of specification text in total, covering a wide range of features on many levels of abstraction. We have therefore found it necessary to further limit our scope; both by leaving out parts that have very little to do with the concurrency and communication aspects that are our primary interest, and by ignoring parts that to some extent duplicate the features that we are already covering. We discuss the chosen omissions in more detail below.

A. Out of scope

All aspects of the AUTOSAR specification that concern representation and selection of data values are beyond the scope of the present study. This includes

- Data types on any level of abstraction, and their associated mappings [2, ch. 5].
- Port interface mapping and data scaling [2, ch. 4.3].
- Measurements and calibration parameters [2, ch. 2.2].
- Variant handling [2, ch. 2.4].

These modeling elements only impose restrictions and demands on the values v and continuations k we reference in our semantics, and which we, in our turn, do not constrain or interpret in any way.

B. Minor variants

The only atomic software component type we include in our formalization is the *Application* software component type. Other atomic component types describe dynamic behaviors that are either just minor variants of the formalized component type (*SensorActuator*, *ComplexDeviceDriver*, *Service*, *ServiceProxy*), or trivial in comparison (*NvBlock*, *EcuAbstraction*, *Parameter*) [2, ch. 3.2.3].

A similar argument justifies our decision to omit port types beyond the sender-receiver and client-server kinds [2, ch. 4.2], as well as per instance memories [2, ch. 7.7]. A port typed by a *TriggerInterface* [2, ch. 4.4.7] is essentially a sender-receiver port with no data contents. Mode management is arguably a much more complex concept in principle, but since the related AUTOSAR mechanism is limited to communication of integer-valued mode requests and acknowledgements, it can be seen as an instance of sender-receiver communication as well [2, ch. 4.4.6]. Per instance memories essentially duplicate the behavior of inter-runnable variables.

Basic Software modules implement behaviors that cannot be captured as ordinary software components due to their dependency on hardware and other platform artifacts [4] [3, ch. 4.2]. However, our abstract treatment of computation dependencies in general confines such differences to one's choice of k continuations, which eliminates the need to formalize Basic Software components through any special means.

C. Left out for brevity

A few areas of the AUTOSAR SWC and RTE specifications have been omitted for the purpose of keeping the size of the resulting formalism at a reasonable level. These are areas where a worked out formalization could prove interesting, and possibly even discover new design ambiguities. However,

we also believe that the techniques required for undertaking such an effort are already demonstrated by our current work, and that making the resulting semantics accessible is an even more important goal than striving for completeness. The major omissions of this kind are:

- Implicit communication, where designated port elements are automatically read/written at the start/end the of runnables that access them [3, ch. 4.3.1.5].
- Category 2 runnables, which declare certain read or receive operations as *WaitPoints* that block progress until the associated events occur [2, ch. 7.2.4.4]. While this form of event handling is fundamentally different from the standard mechanism of activating runnables, it may be noted that the result command that follows from calling a *synchronousServerCallPoint* already defines the essence of such a blocking *WaitPoint*.
- The AUTOSAR COM layer [6] and related events and commands (*DataWriteCompleted*, *DataSendCompleted*, *DataReceiveError*, the *Feedback* command [3, ch. 5.7.5, 5.6.8]). COM is the communication substrate between nodes in a distributed AUTOSAR system and its limited storage capacity introduces an element of resource competition among communicating components. It also leads to the separation of sending and reception into distinct events, as well as additional communication errors.
- Most details found in the *ComSpec* annotations that can be attached to ports (currently only *initValues* are acknowledged) [2, ch. 4.5].
- The meta-programming oriented RTE *Lifecycle* and *Callback* APIs [3, ch. 5.8-9].

VIII. Conclusions and Future Work

In this paper we have provided a formal specification of a substantial core of the AUTOSAR specification. Transition rules and axioms have been defined and linked to the AUTOSAR specification documents, and issues related to choices or ambiguities in the specification have been discussed.

The next step forward for this work is to seek involvement of the AUTOSAR community, in order to gain feedback on our formal interpretation, stimulate clarifying discussions on open design choices, and conceivably also arrive at a consensus regarding the semantics of AUTOSAR as a programming model of its own.

Another line of work is to lift more of the scope limitations listed in Section VII. A formalization of omitted component, port and event types is likely to present very few problems besides an inevitable size increase, which could be mitigated by a compartmentalization of the semantics into independent fragments. More serious work may be required to incorporate the COM layer and the added set of behaviors that follow from component distribution, but the effort would also lead to a significantly better formal coverage of the AUTOSAR specification.

The formal semantics may furthermore provide a foundation for tools that analyze or transform AUTOSAR models. One such example is our AUTOSAR simulator [12], which interprets AUTOSAR models entirely on basis of the transition system, utilizing truly random scheduling and a maximum progress assumption to select among available transitions at each step. The benefit of this simulator is its ability to dynamically execute AUTOSAR Software Component models as they are, without prior commitment to a particular platform or translation into a low level representation. A further avenue of exploration concerns the potential of proving properties of AUTOSAR systems on the basis of their semantics. To prepare for this opportunity, we have defined the transition system used in this paper as a Horn clause file readable by common theorem provers, and, in fact, Appendix A is automatically produced through a simple formatting translation of this file [8].

IX. Acknowledgements

This work was mainly carried out in the project "Resource Aware Functional Programming (RAWFP)" funded by the Swedish Foundation for Strategic Research.

References

[1] AUTOSAR homepage (2016), www.autosar.org
[2] AUTOSAR: AUTOSAR Release 4.2.2 Software Component Template. AUTOSAR consortium (2016)
[3] AUTOSAR: AUTOSAR Release 4.2.2 Specification of RTE. AUTOSAR consortium (2016)
[4] AUTOSAR: Basic Software Module Description Template. AUTOSAR consortium (2016)
[5] AUTOSAR: Explanation of Error Handling on Application Level. AUTOSAR consortium (2016)
[6] AUTOSAR: Specification of AUTOSAR COM. AUTOSAR consortium (2016)
[7] Baeten, J., Middelburg, C.A., Netherlands, E.T.: Process algebra with timing: Real time and discrete time. In: Handbook of Process Algebra. pp. 627–684. Elsevier (2000)
[8] Chalmers AUTOSAR semantics, https://github.com/patrikja/autosar/blob/master/sem/semantics.eprover (2016)
[9] Milner, R.: Communicating and Mobile Systems: the Pi Calculus. Cambridge University Press, New York, NY, USA (1999)
[10] Prasad, K.: A calculus of broadcasting systems. Science of Computer Programming 25(2), 285 – 327 (1995)
[11] Reynolds, J.C.: The discoveries of continuations. Lisp and Symbolic Computation 6(3-4), 233–248 (1993)
[12] The Chalmers AUTOSAR simulator, https://github.com/patrikja/autosar/tree/master/ARSim/ (2016)

APPENDIX

A. Parallel composition

$$p_1 \mid q_1 \xrightarrow{a!d} p_2 \mid q_2 \quad \text{if } p_1 \xrightarrow{a!d} p_2 \text{ and } q_1 \xrightarrow{a?d} q_2 \tag{1}$$

$$p_1 \mid q_1 \xrightarrow{a!d} p_2 \mid q_2 \quad \text{if } p_1 \xrightarrow{a?d} p_2 \text{ and } q_1 \xrightarrow{a!d} q_2 \tag{2}$$

$$p_1 \mid q_1 \xrightarrow{a?d} p_2 \mid q_2 \quad \text{if } p_1 \xrightarrow{a?d} p_2 \text{ and } q_1 \xrightarrow{a?d} q_2 \tag{3}$$

$$p_1 \mid q_1 \xrightarrow{\delta(t)} p_2 \mid q_2 \quad \text{if } p_1 \xrightarrow{\delta(t)} p_2 \text{ and } q_1 \xrightarrow{\delta(t)} q_2 \tag{4}$$

B. Inter-runnable variables

$$\textbf{rinst}(i.r, c, xs, \text{irvRead}(s, k)) \xrightarrow{i.s!\text{IRVR}(v)} \textbf{rinst}(i.r, c, xs, k(v)) \tag{5}$$

$$\textbf{irv}(a, v) \xrightarrow{a?\text{IRVR}(v)} \textbf{irv}(a, v) \tag{6}$$

$$\textbf{rinst}(i.r, c, xs, \text{irvWrite}(s, v, k)) \xrightarrow{i.s!\text{IRVW}(v)} \textbf{rinst}(i.r, c, xs, k(\textbf{void})) \tag{7}$$

$$\textbf{irv}(a, \_) \xrightarrow{a?\text{IRVW}(v)} \textbf{irv}(a, v) \tag{8}$$

C. Exclusive areas

$$\textbf{rinst}(i.r, c, xs, \text{enter}(x, k)) \xrightarrow{i.x!\text{ENT}} \textbf{rinst}(i.r, c, x{:}xs, k(\textbf{void})) \tag{9}$$

$$\textbf{excl}(a, \textbf{false}) \xrightarrow{a?\text{ENT}} \textbf{excl}(a, \textbf{true}) \tag{10}$$

$$\textbf{rinst}(i.r, c, x{:}xs, \text{exit}(x, k)) \xrightarrow{i.x!\text{EX}} \textbf{rinst}(i.r, c, xs, k(\textbf{void})) \tag{11}$$

$$\textbf{excl}(a, \textbf{true}) \xrightarrow{a?\text{EX}} \textbf{excl}(a, \textbf{false}) \tag{12}$$

D. Unbuffered sending/receiving

$$\textbf{rinst}(i.r, c, xs, \text{read}(e, k)) \xrightarrow{i.e!\text{RD}(v)} \textbf{rinst}(i.r, c, xs, k(v)) \tag{13}$$

$$\textbf{delem}(a, \_, v) \xrightarrow{a?\text{RD}(v)} \textbf{delem}(a, \textbf{false}, v) \tag{14}$$

$$\textbf{rinst}(i.r, c, xs, \text{write}(e, v, k)) \xrightarrow{i.e!\text{WR}(v)} \textbf{rinst}(i.r, c, xs, k(\textbf{void})) \tag{15}$$

$$\textbf{delem}(b, \_, \_) \xrightarrow{a?\text{WR}(v)} \textbf{delem}(b, \textbf{true}, v) \quad \text{if } a \Rightarrow b \tag{16}$$

$$\textbf{runnable}(i.r, t, \_, n) \xrightarrow{a?\text{WR}(\_)} \textbf{runnable}(i.r, t, \textbf{pending}, n) \quad \text{if } a \Rightarrow b \text{ and } \textit{DataReceivedEvent}(i.r, b) \tag{17}$$

$$\textbf{rinst}(i.r, c, xs, \text{isUpdated}(e, k)) \xrightarrow{i.e!\text{UP}(u)} \textbf{rinst}(i.r, c, xs, k(u)) \tag{18}$$

$$\textbf{delem}(a, u, v) \xrightarrow{a?\text{UP}(u)} \textbf{delem}(a, u, v) \tag{19}$$

$$\textbf{rinst}(i.r, c, xs, \text{invalidate}(e, k)) \xrightarrow{i.e!\text{INV}} \textbf{rinst}(i.r, c, xs, k(\textbf{void})) \tag{20}$$

$$\textbf{delem}(b, \_, \_) \xrightarrow{a?\text{INV}} \textbf{delem}(b, \textbf{true}, \textbf{invalid}) \quad \text{if } a \Rightarrow b \tag{21}$$

E. Buffered sending/receiving

$$\textbf{rinst}(i.r, c, xs, \text{receive}(e, k)) \xrightarrow{i.e!\text{RCV}(v)} \textbf{rinst}(i.r, c, xs, k(v)) \tag{22}$$

$$\textbf{qelem}(a, n, v{:}vs) \xrightarrow{a?\text{RCV}(v)} \textbf{qelem}(a, n, vs) \tag{23}$$

$$\textbf{qelem}(a, n, \epsilon) \xrightarrow{a?\text{RCV}(\textbf{nodata})} \textbf{qelem}(a, n, \epsilon) \tag{24}$$

$$\textbf{rinst}(i.r, c, xs, \text{send}(e, v, k)) \xrightarrow{i.e!\text{SND}(v,\epsilon)} \textbf{rinst}(i.r, c, xs, k(\textbf{ok})) \tag{25}$$

$$\textbf{rinst}(i.r, c, xs, \text{send}(e, v, k)) \xrightarrow{i.e!\text{SND}(v,as)} \textbf{rinst}(i.r, c, xs, k(\textbf{limit})) \tag{26}$$
if $as \neq \epsilon$ and $i.e \Rightarrow a$ for all $a \in as$

$$\textbf{qelem}(b, n, vs) \xrightarrow{a?\text{SND}(v,as)} \textbf{qelem}(b, n, vs{:}v) \tag{27}$$
if $a \Rightarrow b$ and $|vs| < n$ and $b \notin as$

$$\textbf{qelem}(b, n, vs) \xrightarrow{a?\text{SND}(\_,as)} \textbf{qelem}(b, n, vs) \tag{28}$$
if $a \Rightarrow b$ and $|vs| = n$ and $b \in as$

$$\textbf{runnable}(i.r, t, \_, n) \xrightarrow{a?\text{SND}(\_,as)} \textbf{runnable}(i.r, t, \textbf{pending}, n) \tag{29}$$
if $a \Rightarrow b$ and *DataReceivedEvent*$(i.r, b)$ and $b \notin as$

$$\textbf{runnable}(i.r, t, act, n) \xrightarrow{a?\text{SND}(\_,as)} \textbf{runnable}(i.r, t, act, n) \tag{30}$$
if $a \Rightarrow b$ and *DataReceivedEvent*$(i.r, b)$ and $b \in as$

F. Calling a server

$$\textbf{rinst}(i.r, c, xs, \text{call}(o, v, k)) \xrightarrow{i.o!\text{CALL}(\_,v)} \textbf{rinst}(i.r, c, xs, \text{result}(o, k)) \tag{31}$$
if *synchronousServerCallPoint*$(i.r, i.o)$

$$\textbf{rinst}(i.r, c, xs, \text{call}(o, v, k)) \xrightarrow{i.o!\text{CALL}(\_,v)} \textbf{rinst}(i.r, c, xs, k(\textbf{ok})) \tag{32}$$
if *asynchronousServerCallPoint*$(i.r, i.o)$

$$\textbf{oper}(a, m, \textbf{done}(\_)) \xrightarrow{a?\text{CALL}(m,v)} \textbf{oper}(a, m, \textbf{calling}(t)) \tag{33}$$
if *serverCallPointTimeout*(a, t)

$$\textbf{runnable}(i.r, t, cs, n) \xrightarrow{a?\text{CALL}(m,v)} \textbf{runnable}(i.r, t, cs{:}(a, m, v), n) \tag{34}$$
if *OperationInvokedEvent*$(i.r, b)$ and $b \Rightarrow a$

$$\textbf{rinst}(i.r, c, xs, \text{call}(o, v, k)) \xrightarrow{i.o!\text{BUSY}} \textbf{rinst}(i.r, c, xs, k(\textbf{limit})) \tag{35}$$

$$\textbf{oper}(a, m, \textbf{calling}(t)) \xrightarrow{a?\text{BUSY}} \textbf{oper}(a, m, \textbf{calling}(t)) \tag{36}$$

$$\textbf{runnable}(i.r, t, cs, n) \xrightarrow{a?\text{BUSY}} \textbf{runnable}(i.r, t, cs, n) \tag{37}$$
if *OperationInvokedEvent*$(i.r, b)$ and $b \Rightarrow a$

G. Passing back a server result

$$\textbf{rinst}(i.r, c, xs, \text{result}(o, k)) \xrightarrow{i.o!\text{RES}(v)} \textbf{rinst}(i.r, c, xs, k(v)) \tag{38}$$
if *synchronousServerCallPoint*$(i.r, i.o)$
and $v \neq \textbf{nodata}$

$$\textbf{rinst}(i.r, c, xs, \text{result}(o, k)) \xrightarrow{i.o!\text{RES}(v)} \textbf{rinst}(i.r, c, xs, k(v)) \tag{39}$$
if *asynchronousServerCallPoint*$(i.r, i.o)$

$$\textbf{oper}(a, m, \textbf{calling}(t)) \xrightarrow{a?\text{RES}(\textbf{nodata})} \textbf{oper}(a, m, \textbf{calling}(t)) \tag{40}$$

$$\textbf{oper}(a, m, \textbf{done}(v)) \xrightarrow{a?\text{RES}(v)} \textbf{oper}(a, m, \textbf{done}(v)) \tag{41}$$

$$\textbf{rinst}(a, (b, m, \_), xs, \text{return}(v)) \xrightarrow{b!\text{RET}(m,v)} \textbf{rinst}(a, \textbf{void}, xs, \text{return}(\textbf{void})) \tag{42}$$

$$\textbf{oper}(a, m, \textbf{calling}(\_)) \xrightarrow{a?\text{RET}(m,v)} \textbf{oper}(a, m+1, \textbf{done}(v)) \tag{43}$$

$$\textbf{oper}(a, m, \textbf{calling}(0)) \xrightarrow{a!\text{RET}(m,\textbf{timeout})} \textbf{oper}(a, m+1, \textbf{done}(\textbf{timeout})) \tag{44}$$

$$\textbf{runnable}(i.r, t, \_, n) \xrightarrow{a?\text{RET}(\_,\_)} \textbf{runnable}(i.r, t, \textbf{pending}, n) \quad \text{if } \textit{asynchronousServerCallReturnsEvent}(i.r, a) \tag{45}$$

$$\textbf{rinst}(a, (b, m, \_), xs, \text{return}(v)) \xrightarrow{b!\text{SKIP}(m)} \textbf{rinst}(a, \textbf{void}, xs, \text{return}(v)) \tag{46}$$

$$\textbf{oper}(a, m, srv) \xrightarrow{a?\text{SKIP}(n)} \textbf{oper}(a, m, srv) \quad \text{if } m \neq n \tag{47}$$

$$\textbf{runnable}(i.r, t, act, n) \xrightarrow{a?\text{SKIP}(\_)} \textbf{runnable}(i.r, t, act, n) \quad \text{if } \textit{asynchronousServerCallReturnsEvent}(i.r, a) \tag{48}$$

H. Spawning and terminating

$$\textbf{runnable}(a, 0, \textbf{pending}, n) \xrightarrow{a!\text{NEW}} \textbf{runnable}(a, t, \textbf{idle}, n+1) \mid \textbf{rinst}(a, \textbf{void}, \epsilon, k(\textbf{void})) \quad \text{if } (n = 0 \vee \textit{canBeInvokedConcurrently}(a)) \text{ and } \textit{minimumStartInterval}(a, t) \text{ and } \textit{implementation}(a, k) \tag{49}$$

$$\textbf{runnable}(a, 0, (b, m, v){:}cs, n) \xrightarrow{a!\text{NEW}} \textbf{runnable}(a, t, cs, n+1) \mid \textbf{rinst}(a, (b, m, v), \epsilon, k(v)) \quad \text{if } (n = 0 \vee \textit{canBeInvokedConcurrently}(a)) \text{ and } \textit{minimumStartInterval}(a, t) \text{ and } \textit{implementation}(a, k) \tag{50}$$

$$\textbf{rinst}(a, \textbf{void}, \epsilon, \text{return}(\_)) \xrightarrow{a!\text{TERM}} \textbf{0} \tag{51}$$

$$\textbf{runnable}(a, t, act, n) \xrightarrow{a?\text{TERM}} \textbf{runnable}(a, t, act, n-1) \tag{52}$$

I. Passing time

$$\textbf{rinst}(a, c, xs, code) \xrightarrow{\delta(\_)} \textbf{rinst}(a, c, xs, code) \tag{53}$$

$$\textbf{excl}(a, v) \xrightarrow{\delta(\_)} \textbf{excl}(a, v) \tag{54}$$

$$\textbf{irv}(a, v) \xrightarrow{\delta(\_)} \textbf{irv}(a, v) \tag{55}$$

$$\textbf{qelem}(a, n, vs) \xrightarrow{\delta(\_)} \textbf{qelem}(a, n, vs) \tag{56}$$

$$\textbf{delem}(a, u, v) \xrightarrow{\delta(\_)} \textbf{delem}(a, u, v) \tag{57}$$

$$\textbf{oper}(a, m, \textbf{done}(v)) \xrightarrow{\delta(\_)} \textbf{oper}(a, m, \textbf{done}(v)) \tag{58}$$

$$\textbf{oper}(a, m, \textbf{calling}(v)) \xrightarrow{\delta(t)} \textbf{oper}(a, m, \textbf{calling}(v-t)) \quad \text{if } t \leq v \tag{59}$$

$$\textbf{runnable}(a, v, act, n) \xrightarrow{\delta(t)} \textbf{runnable}(a, v-t, act, n) \quad \text{if } t \leq v \tag{60}$$

$$\textbf{runnable}(a, 0, act, n) \xrightarrow{\delta(\_)} \textbf{runnable}(a, 0, act, n) \tag{61}$$

$$\textbf{timer}(a, t_p, v) \xrightarrow{\delta(t)} \textbf{timer}(a, t_p, v-t) \quad \text{if } t \leq v \tag{62}$$

$$\textbf{timer}(a, t_p, 0) \xrightarrow{a!\text{TICK}} \textbf{timer}(a, t_p, t_p) \tag{63}$$

$$\textbf{runnable}(a, t, \_, n) \xrightarrow{a?\text{TICK}} \textbf{runnable}(a, t, \textbf{pending}, n) \tag{64}$$

$$\textbf{rinst}(a, c, xs, k) \xrightarrow{-?-} \textbf{rinst}(a, c, xs, k) \tag{65}$$

$$\textbf{timer}(a, t_p, t) \xrightarrow{-?-} \textbf{timer}(a, t_p, t) \tag{66}$$

$$\textbf{excl}(b, v) \xrightarrow{a?-} \textbf{excl}(b, v) \quad \text{if } a \neq b \tag{67}$$

$$\textbf{irv}(b, v) \xrightarrow{a?-} \textbf{irv}(b, v) \quad \text{if } a \neq b \tag{68}$$

$$\textbf{oper}(b, m, srv) \xrightarrow{a?-} \textbf{oper}(b, m, srv) \quad \text{if } a \neq b \tag{69}$$

$$\textbf{qelem}(b, n, vs) \xrightarrow{a?-} \textbf{qelem}(b, n, vs) \quad \text{if } a \neq b \text{ and } a \not\Rightarrow b \tag{70}$$

$$\textbf{delem}(b, u, v) \xrightarrow{a?-} \textbf{delem}(b, u, v) \quad \text{if } a \neq b \text{ and } a \not\Rightarrow b \tag{71}$$

$$\textbf{runnable}(i.r, t, act, n) \xrightarrow{a?-} \textbf{runnable}(i.r, t, act, n) \tag{72}$$
$$\text{if } a \neq i.r$$
$$\text{and } \neg(a \Rightarrow b \wedge \textit{DataReceivedEvent}(i.r, b))$$
$$\text{and } \neg(\textit{OperationInvokedEvent}(i.r, b) \wedge b \Rightarrow a)$$
$$\text{and } \neg(\textit{asynchronousServerCallReturnsEvent}(i.r, a))$$