A Formal Operational Model of ACT-R: Structure and Behaviour

Vincent Langenfeld (langenfv@informatik.uni-freiburg.de)
Department of Computer Science, University of Freiburg

Bernd Westphal (westphal@informatik.uni-freiburg.de)
Department of Computer Science, University of Freiburg

Andreas Podelski (podelski@informatik.uni-freiburg.de)
Department of Computer Science, University of Freiburg

Abstract

It is a long standing challenge to devise a formal model of ACT-R as a basis for formal reasoning on ACT-R. The ACT-R architecture is a composition of components (such as modules) with predefined interfaces between components and predefined interactions on the interfaces. Reasoning over the correctness of a formal model of ACT-R benefits from the separation of abstraction levels i.e. reasoning on the level of interfaces and interactions between components in isolation from the concrete behaviour of each component.
We propose a formal semantics of ACT-R that preserves the structural properties of architectural components, i.e., the interfaces of modules to the remaining architecture as well as communication between modules within the architecture. We demonstrate how our new formal semantics of ACT-R serves to prove the correctness of the timed automaton based operational semantics for ACT-R (TA-ACT-R) on the level of architectural components.

Keywords: ACT-R; Cognitive Architecture; Formal Operational Semantic;

Introduction

The cognitive architecture ACT-R (Anderson, 2007) is widely used to model and validate psychological theories. To do so, the psychological theory is realised as a cognitive model and executed on the ACT-R architecture, with the subsequent evaluation of statistical data.

Commonly, the ACT-R architecture is seen as *one* parameterised architecture that is used to execute cognitive models. This view is reinforced by the ACT-R tool (Bothell, 2013), which allows us to enable or disable certain modules and to influence the simulation through certain numerical parameters (like activation decay value). Going back to the ACT-R theory (Anderson, 2007), we see that ACT-R is actually defined as a whole *family* of architectures in the previous sense. Following Anderson, an architecture is a composition of modules that each model a self-contained aspect of human cognition such as memory, or a function of perception. What these modules have in common is that they interact exclusively through an interface that is described in (Anderson, 2007). The interface consists of working instructions (called actions) and shared memory (called buffers) with additional restrictions, e.g., that one buffer can only contain one chunk at a time. The executions of a cognitive model on a given architecture emerge from interactions between the production rules in the cognitive model and the composition of modules in the considered architecture, and the interplay between the modules: Production rules can access modules' buffers in order to check whether their precondition is satisfied, that is, whether the rule is enabled. If a production rule is enabled, it can be executed, that is, it requests actions of the modules in the architecture.

It is a long standing challenge to devise a formal, mathematical model of the ACT-R architecture and the behaviour of cognitive models on an adequate level of abstraction, because (Anderson, 2007) only provides an informal description. As of today, many details are clarified by the implementation in the ACT-R tool (Bothell, 2013), that is, the implementation effectively provides a precise ACT-R semantics (cf. Figure 1 on page 2). A formal, mathematical model of ACT-R in contrast would allow to *prove* that the ACT-R tool (and re-implementations in other programming languages) all correctly implement the ACT-R theory. Quite a number of approaches to define ACT-R formally exist. There are earlier works like (Stewart & West, 2007), which was designed to support the development of a new simulation tool, or (Schultheis, 2009) for the investigation of complexity theory questions. In recent years, a branch of research emerged that is driven by the need to exhaustively analyse cognitive models for certain kinds of modelling errors (Gall & Frühwirth, 2014; Gall & Frühwirth, 2018; Langenfeld, Westphal, Albrecht, & Podelski, 2018). These works are based on a formal operational semantics F-ACT-R (Albrecht & Ragni, 2014) (and its later refinement in (Gall & Frühwirth, 2018)). What these works have in common is that they focus on the behaviour of cognitive models and abstract the cognitive architecture into one monolithic unit (cf. Figure 1). Recent research like (Langenfeld, Westphal, & Podelski, 2019) has shown the scalability of exhaustive (symbolic) analysis and simulation of architectures and models (here executed on the Timed Automaton-based architecture model TA-ACT-R). Yet proving correctness of such models with regard to a monolithic formalisation such as F-ACT-R (Albrecht & Westphal, 2014) turns out to be difficult, as the proof has to mix different abstraction levels, i.e. the proof has to regard not only the behaviour of all components of the architecture at their interfaces but also the (possibly complex) behaviour of the components itself. Such proof would greatly benefit from decomposition (or rather compositionality) of a formalisation. Hence there is a need for a formal semantics of ACT-R that highlights the structural properties of the architecture and explicitly formulates the expected behaviour of each component as stated in (Anderson, 2007). An

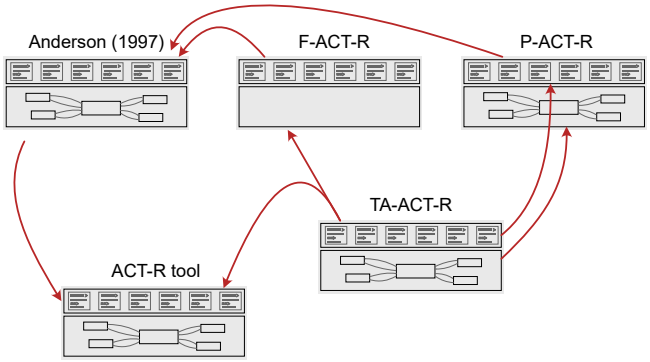


Figure 1: The different models of the ACT-R Theory in relation. For each model of ACT-R represented by a grey background, the upper rectangle represents a cognitive model and the lower rectangle represents the architecture.

additional benefit is that such a modular formal semantics of ACT-R can also be well-suited for architecture research, that is, the investigation of modules beyond the standard set, and a rigorous investigation of the interfaces and the module interaction on an abstract level, as the behaviour of a model is largely dependent on the interplay of the different modules within the architecture. Other works such as (Ragni et al., 2018) use a different level of abstraction as they aim at a generalisation of in particular (Albrecht & Ragni, 2014) into a general cognitive architecture that is supposed to abstractly model the computations of different cognitive architectures (such as ACT-R or SOAR) in order to compare, e.g., the outcome of cognitive models on different architectures.

In this work, we propose a formal semantics of ACT-R that preserves the structural properties of architectural components, i.e., the interfaces of modules to the remaining architecture as well as communication between modules within the architecture. We uses a formalism that is inspired by process calculi. The idea of the new formal model is that each module and each rule becomes a process with a well-defined interface. The interfaces define which information is accessed by (or shared with) other parts and the allowed interactions. The behaviour of a cognitive model on a particular architecture is then defined by the parallel composition of the rule and module processes. To demonstrate the usefulness of our formalisation, we outline how to show that TA-ACT-R is a correct implementation (or refinement) of our process-based semantics (and thus the ACT-R cognitive architecture) by showing that TA-ACT-R correctly implements the given interfaces and interactions. On the example of TA-ACT-R, the benefits of the P-ACT-R view for architectural research become particularly evident: Showing that a newly proposed module is a proper ACT-R model amounts to showing that it implements the general module interface and the abstract cycle of action processing (cf. Figure 1).

Preliminaries

Processes are defined over a set Chan of channels (on which processes can synchronise) and a set Var of variables (which processes can read or write). A **process** is a tuple $\pi = ((I,V),(O,W))$, with **incoming channels** $I \subseteq \mathsf{Chan}$ and **outgoing channels** $O \subseteq \mathsf{Chan}$. A process offers a set of **outgoing variables** $W \subseteq \mathsf{Var}$ to be read by other processes, and requires a set of **incoming variables** $V \subseteq \mathsf{Var}$ to be readable by itself. Intuitively, the channels allow to express that information from another process is required (thus waiting for a process sending on a certain channel) or that the continuation of a computation by another process is required (thus waiting for a process receiving on a certain channel). Additional parameters can be exchanged over the shared variables when a synchronisation happens (and only then).

Each process has a set C of **configurations**. A configuration $c \in C$ provides a value for each outgoing variable $w \in W$, and may include any number of other information. We write $c[\![w]\!]$ for the value of w in c.

Two configurations c, c' of a process are in a **transition relation** if the process is synchronising on a channel, either by receiving or sending, or if the process is waiting without any synchronisation. Transitions depend on the values of the incoming variables, these values are provided by an environment ε. Processes can have three kinds of transitions from configuration c to successor configuration c': (i) The process receives a synchronisation on an incoming channel $\alpha \in I$; these transitions we write as $[\varepsilon]\langle c\rangle \xrightarrow{\alpha?} c'$, (or $c \xrightarrow{\alpha?}_{\varepsilon} c'$ for short), (ii) The process initiates a synchronisation on an outgoing channel $\alpha \in O$, denoted; these transitions we write as $[\varepsilon]\langle c\rangle \xrightarrow{\alpha!} c'$, (or $c \xrightarrow{\alpha!}_{\varepsilon} c'$ for short). (iii) A process can also wait for duration $t \in \mathbb{R}_0^+$ as a transition; these transitions we write as $c \xrightarrow{t} c'$.

The **parallel composition** of a set of processes $\pi_1 \| \ldots \| \pi_n$ has configurations $(c_1,\ldots,c_n)$, with c_i being configuration of process π_i, $1 \leq i \leq n$. There are two kinds of transitions between such configurations. There is a transition $(c_1,\ldots,c_n) \xrightarrow{\alpha} (c'_1,\ldots,c'_n)$ if $[c_1,\ldots,c_{i-1},c_{i+1},\ldots,c_n]\langle c_i\rangle \xrightarrow{\alpha!} c'$, $\alpha \in O_i$ and $[c_1,\ldots,c_{j_k-1},c_{j_k+1},\ldots,c_n]\langle c_{j_k}\rangle \xrightarrow{\alpha?} c'$, $\alpha \in I_{j_k}$, i.e., one sender and all receivers on channel α do a joint transition while the configurations of all other processes form the environment (i.e., offer values) for the input variables. Alternatively, all processes can wait simultaneously. There is a transition $(c_1,\ldots,c_n) \xrightarrow{t} (c'_1,\ldots,c'_n)$ if $c_i \xrightarrow{t} c'_i$ for all $1 \leq i \leq n$.

The ACT-R Process Semantic (P-ACT-R)

In this section, we propose a new compositional operational semantics for the ACT-R architecture with an explicit representation of structural properties. The main idea is to view a cognitive architecture as a parallel composition of processes, one for the procedural and one for each other module, and to view a cognitive model as a parallel composition of finitely many rules with a compatible cognitive architecture. Each process has a well-defined interface (which consists of input and output channels, and sets of variables that are read or written

by other processes) that defines its structural view, and a set of possible transitions that define its behaviour. The emphasis of the P-ACT-R formalisation lies on the possible interactions between components. The architecture hence abstracts from all computations inside the modules that are not necessary to describe these possible interactions, for example, the particular computation of rule enabledness. In the section on TA-ACT-R below, we outline how it is possible to refine these aspects of the P-ACT-R semantics into concrete computations.

Chunks

Let $\mathcal{C}$ be a possibly infinite set of chunks including the null chunk $\perp$. The null chunk is used by actions that do not pass any ordinary chunk (see below). Note that, in ACT-R process semantics (similar to (Ragni et al., 2018)), we abstract from the complex data structure of a chunk (including slots and their values) hence chunks in $\mathcal{C}$ can be seen as unique chunk identities that are identified with the concrete, structured chunk that they denote.

Module

The following Definition 1 provides an abstract formalisation of ACT-R modules as processes. Recall that, in ACT-R, a module models a self contained cognitive process, such as access to memory or the visual apparatus. Modules communicate with the remainder of the ACT-R architecture and the rules over buffers that offer information in the form of chunks, and status information in the form of boolean flags (called queries) to the architecture. A module can be triggered to perform a so-called action, like remembering or shifting the visual attention. In ACT-R, a module does offer a set of discrete actions, e.g., to remember something. In many cases, arguments are added to the action e.g., to remember something that is blue. The result of an action as well as the (possibly instantaneous) delay produced by performing the action, is up to the implementation of the module.

Definition 1 (P-ACT-R Module)**.** *A module in the ACT-R process semantics is a process* $m = ((I,V),(O,W))$ *with* $I = \{\tau^m, a_1^m(\gamma), \ldots, a_n^m(\gamma), harvest(\gamma) \mid \gamma \in \mathcal{C}\}$, $V = \emptyset$, $O = \{bc, harvest(\gamma) \mid \gamma \in \mathcal{C}\}$, *and* $W = \{B^m, Q^m\}$, *whose set of configurations is partitioned into* stable *and* unstable *configurations, and whose behaviour always has one of the following two forms:*

$$c \xrightarrow{50} c_0 \xrightarrow{\alpha?} c_1 \xrightarrow{t} c' \tag{1}$$

$$c \xrightarrow{50} c_0 \xrightarrow{\alpha?} c_1 \xrightarrow{t} c_2 \xrightarrow{bc!} c' \tag{2}$$

Here, c, c' *are stable and* c_i *are unstable configurations.*

Actions are modelled in P-ACT-R by a set of channels $a_1^m(\gamma), \ldots, a_n^m(\gamma)$. Executing a buffer action (for example, to remember something) is modelled by synchronising on the according channel with all arguments of the action passed as a chunk γ. The incoming channel $harvest(\gamma)$ is to be able to receive a chunk γ from other modules, e.g., as part of a fact learning mechanism. The action τ^m is a special action that can be sent to the module in order to request that the module does not do anything. The outgoing channels *bc* and *harvest* are channels that are internal to the architecture. Channel *bc* is used to inform the procedural module (see below) of the fact that an action with non zero delay has ended. Channel *harvest* is used to announce a chunk to the processes in the architecture to realise ACT-R's harvesting concept (Anderson, 2007). Buffers are modelled by the output variables B^m with type $\mathcal{C}$ and buffer queries by the output variables Q^m with boolean type. Note, that we consider B^m to be exactly one buffer for simplicity. In Definition 1, Sequence (1) models the case where the module waits for an action, executes the action (followed by a possibly zero delay). Sequence (2) models the case where, after some time, the end of the action is communicated to the procedural system via the *bc* channel.

Note, that the *harvest* action is not part of the transition sequences for simplicity. It may appear as a synchronisation between two modules before or after an action (including τ^m).

Procedural

The following Definition 2 provides an abstract formalisation of ACT-R procedural. Recall that, he procedural is responsible for choosing and executing production rules. After a waiting period that shall model the time spent on selecting a production rule, the procedural evaluates the precondition of all production rules in the current cognitive state. Amongst the rules whose precondition is fulfilled, the procedural chooses one rule, and executes its action. If no rule is applicable because no precondition is fulfilled, the procedural waits for the cognitive state to change e.g. due to a module finishing its action.

Contrary to the common view that the procedural is also a module, we define the procedural on its own, because neither its interface nor its behaviour matches that of modules.

Definition 2 (P-ACT-R Procedural)**.** *Let* $\mathcal{R}$ *be the set of production rule identities. A procedural is a process* $p = ((I,V),(O,W))$ *with* $I = \{bc\}$, $V = \{E^r \mid r \in \mathcal{R}\}$, $O = \{cr, fire(r) \mid r \in \mathcal{R}\}$ *and* $W = \emptyset$ *whose set of configurations is partitioned into* stable *and* unstable *configurations, and whose behaviour always has one of the following two forms:*

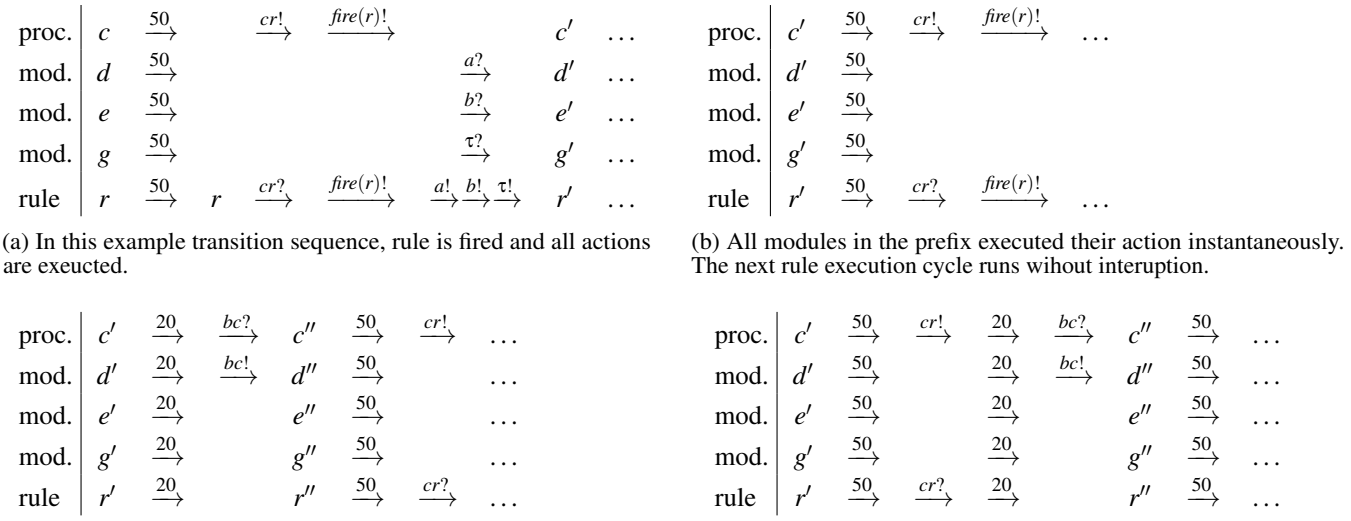
$$c \xrightarrow{50} c_0 \xrightarrow{cr!} c_1 \xrightarrow{fire(r)!} c' \tag{3}$$

$$c \xrightarrow{50} c_0 \xrightarrow{cr!} c_1 \xrightarrow{t} c_2 \xrightarrow{bc?} c' \tag{4}$$

$$c \xrightarrow{t} c_0 \xrightarrow{bc?} c' \qquad \text{if } t \leq 50 \tag{5}$$

Here c, c' *are stable and* c_i *are unstable configurations.*

The procedural has only one incoming channel *bc*, which is used by modules to inform the procedural that an action has finished and thus the configuration of the module may have changed. To choose a rule for execution, the procedural can access if the precondition of a production rule r was satisfied by reading the variable E^r. Also, in contrast to ACT-R where the procedural is responsible for checking the precondition and execution of the action of a rule, in the P-ACT-R the procedural is only responsible for scheduling. In other words, the



(a) In this example transition sequence, rule is fired and all actions are exeucted.

(b) All modules in the prefix executed their action instantaneously. The next rule execution cycle runs wihout interuption.

(c) The next rule execution cycle starts, but a module finishes its action after another 20-delay. A new execution cycle is started afterwards.

(d) The next execution cycle starts, but no precondition of a production rule is fulfilled. After a 70-delay a module finishes its action. A new execution cycle is started afterwards.

Figure 2: Example transition sequences of the P-ACT-R semantic. All sequences begin with a prefix (Figure 2a), and extend the transition sequence by the different possible cases of interaction between procedural, an exemplary production rule and modules.

P-ACT-R procedural synchronises with all production rules on *cr* to let the production rules check their precondition, and it synchronises with one production rule on $fire(r)$ to let the rule execute its action. Sequence (3) models the basic cycle of waiting, letting all production rules check their preconditions, and firing one enabled rule. Sequence (4) models the case, in which after the check no rule can fire and the procedural waits for the cognitive state to change (i.e. a module has ended its computation and synchronises on *bc*) (5) models the case in which a module changes the cognitive state before a rule is chosen and executed, causing the rule execution cycle to start from the beginning.

Cognitive Architecture

We define a **cognitive architecture** in ACT-R process semantics as the parallel composition $p\|m_1\|\ldots\|m_n$ of a procedural p and finitely many modules $m_1,\ldots,m_n$.

Note that the channels *bc* and $harvest(\gamma)$ are only used for the communication between modules and procedural resp. only modules (i.e., inside the architecture) and could thus be hidden from the rules.

Production Rule

The following Definition 3 provides an abstract formalisation of ACT-R cognitive models. A cognitive model in ACT-R consists of a set of production rules modelling the researched cognitive task. Production rules have two parts: A precondition and an action. The precondition is a proposition over all buffers and buffer queries that defines when the rule is enabled. A rule action (or action for short) is a set of module actions with parameters, at most one per module in the architecture. A rule action could, for example, consist of the 'remember' action of the declarative module with a parameter that says 'of colour blue', and the 'clear' action of the imaginal module.

Definition 3 (P-ACT-R Production Rule)**.** *A production rule is a process* $r=((I,V),(O,W))$ *with* $I=\{cr,fire(r)\}$, $V=\{B^r_1,\ldots,B^r_{n_r},Q^r_1,\ldots,Q^r_{m_r}\}$, $O=\{a^r_1,\ldots,a^r_{n_r}\}$ *and* $W=\{E^r\}$. *whose set of configurations is partitioned into* stable *and* unstable *configurations, and whose behaviour always has one of the following two forms:*

$$c \xrightarrow{t} c \tag{6}$$

$$c \xrightarrow{cr?} c' \qquad \text{if } c'[\![E^r]\!]=0 \tag{7}$$

$$c \xrightarrow{cr?} c_1 \xrightarrow{fire(r)?} c_2 \xrightarrow{a^r_{n_0}!} \cdots \xrightarrow{a^r_{n_k}!} c' \qquad \text{if } c'[\![E^r]\!]=1 \tag{8}$$

Here, c,c' *are stable and* c_i *are unstable configurations.*

After synchronising on the *cr* channel, a production rule may receive a message on the $fire(r)$ channel, if $c'[\![E^r]\!]=1$. The concrete semantics of the check is not part of the P-ACT-R formalisation. After a synchronisation on the $fire(r)$ channel, an action for each module in the architecture is executed. If no specific action is required of a module, τ^m is used. Sequence (6) models waiting for other processes, sequence (7) models if the precondition is not fulfilled and sequence (8) models if the precondition is fulfilled and the rule is fired.

Cognitive Model

We define a **cognitive model** in ACT-R process semantics as the parallel composition of an architecture with production rules r_1 to r_M, with all interfaces being satisfied, i.e., for each input variable v of a rule, there is a unique module that provides the variable as an output (in W). For each output channel of a rule in O, there is a unique module that provides it as part of its incoming channels I, and the output channels of

each rule include the internal action τ^m of each module of the architecture.

The computations of a cognitive model consist of transition sequences where $fire(r)$ is followed by exactly N actions, one for each module in the process model (and hence one for each buffer). An example of possible computations resulting from different delays of modules are depicted in Figure 2. A cognitive model configuration $(c_p, c_{m_1}, \ldots, c_{m_N}, c_{r_1}, \ldots, c_{r_M})$ is stable if and only if all components are stable

A stable state models a configuration of the module (and other processes) that is relatable to cognitive states in the execution of an ACT-R model (on the ACT-R architecture). Unstable states make the inner workings of the architecture and the causes of executions visible.

TA-ACT-R: A concrete instance of P-ACT-R

The process-based model P-ACT-R from the previous section is meant to be a theoretical formal model of the concepts from (Anderson, 2007) on a directly corresponding level of abstraction. To this end, P-ACT-R aims at having a one-to-one relation between activities described in (Anderson, 2007) (such as checking rules for enabledness, delay, notifying the procedural of module changes, etc.) and transitions in the execution of a cognitive model on a cognitive architecture in P-ACT-R. P-ACT-R in particular abstracts from the syntax of rules' premises and actions and from the way *how* the enabledness check is conducted for the premise of a given rule and *how* actions are interpreted by modules.

In this section, we show that a concrete, simulatable, verifiable model of cognitive ACT-R models (in the sense of the ACT-R tool (Bothell, 2013)) that uses the formalism of timed automata and for which we can make plausible that it is a proper model (Langenfeld et al., 2019), is in particular an instance of the P-ACT-R model of ACT-R-like cognitive architectures and models in general. In other words, the TA-ACT-R model that we introduce in the following *correctly* realises rule-based cognitive reasoning as specified by the process-based P-ACT-R model.

This section is structured as follows. For self-containedness, we briefly recall the definition of timed automata. In the following subsections, we outline a construction procedure for timed automata for the procedural, for modules, and for rules given an ACT-R model (e.g., in form of an input file of the ACT-R tool (Bothell, 2013)). In the last subsections, we recall the notion of a weak (bi)simulation and outline how to prove that there is such a weak bisimulation relation between networks of timed automata constructed according to the TA-ACT-R procedure and the P-ACT-R model for the same cognitive model.

Timed Automata

Timed Automata (TA) (Alur & Dill, 1994) are a formal operational model of real-time systems. In its simplest case, a **timed automaton** is a tuple $\mathcal{A} = (L, B, V, X, I, E, \ell_{init})$ with L being a finite set of locations (including the initial location ℓ_{init}), a set of channels B, a set of variables V, and a set of clocks X, a function I labelling each location with a clock constraint (called an invariant), and a finite set of edges E. An edge is a tuple $(\ell, \alpha, g, \rho, \ell')$ with source location ℓ and destination ℓ', action α (being either an internal action τ, output $b!$ or input $b?$ on a channel $b \in B$), a clock constraint g as a guard, and an update $\rho \subseteq X$ of clocks to reset.

The operational semantics of a **network of timed automata** $\mathcal{N} = \mathcal{A}_1 \parallel \ldots \parallel \mathcal{A}_n$ (with $\parallel$ denoting parallel composition) is a labelled transition system over **configurations** $\langle \vec{\ell}, \nu \rangle$ with $\vec{\ell}_i$ being a location of $\mathcal{A}_i$ and $\nu : X \to \mathbb{R}_0^+$ being a valuation of clocks. Two configurations are in a transition relation $\langle \vec{\ell}, \nu \rangle \xrightarrow{\lambda} \langle \vec{\ell}', \nu' \rangle$ iff $\lambda \in \mathbb{R}_0^+$, $\vec{\ell} = \vec{\ell}'$ and $\nu' = \nu + \lambda$ satisfy the invariants of $\vec{\ell}$ (delay transition), or there is an edge $(\ell, \alpha, g, \rho, \ell') \in E_i$ so that $\vec{\ell}_i = \ell$ and $\vec{\ell}'_i = \ell'$, g is satisfied by ν and ν' by resetting all clocks in ρ (internal transition), or there are two edges enabled in two different automata in $\mathcal{N}$ with complementary input and output actions (rendezvous transition). A **computation path** is a sequence of configurations starting with $\langle \vec{\ell}_0, \nu_0 \rangle$ with $\vec{\ell}_0$ being the set of all initial locations, ν_0 has all clocks at zero, and all subsequent transitions are in a transition relation.

We use a graphical representation of timed automata, where the double outline location is initial, and locations with a superscript 'C' are committed locations, which inhibit delay. Invariants at locations (if not equal to true) are shown in purple. Edges are annotated with actions (in orange), guards (in green), and updates (in blue).

Chunks

In ACT-R a chunk is seen as a number of slots which themselves can only contain other chunks. The leaves of the resulting tree are chunks without any slots themselves. In the timed automata model, we implement the concept of chunk identities and have (in the set of variables V) a lookup table that is indexed by chunk identities where each entry contains slot/value pairs. As most operations in the ACT-R architecture work by value, i.e., slot contents of buffers in the architecture are copied to other buffers, chunk identity lookups are only necessary in the case of a dereference action, and during interaction with declarative memory.

Modules

The TA model of a module m is a timed automaton $\mathcal{A}_m = (L, B, X, V, I, E, \ell_{ini})$ with channels $B = \{a_1^m, \ldots, a_n^m, bc, done, harvest\}$, variables $V = \{b_m, b_q, b^v, b^{op}\}$, and the locations and actions as shown in Figure 3. The schematic automaton includes one example of an action where the result is available after a delay, and an example where the result is available instantaneously.

Recall that the P-ACT-R semantics summarises module actions into one process action with a chunk parameter. The timed automata formalism does not support this brevity hence we implement module actions with a combination of channels $a_1^m, \ldots, a_n^m$, a variable b^v that contains arguments for the action and another (optional) variable b^{op} that contains operators. Compared to a cognitive model in the input language of the

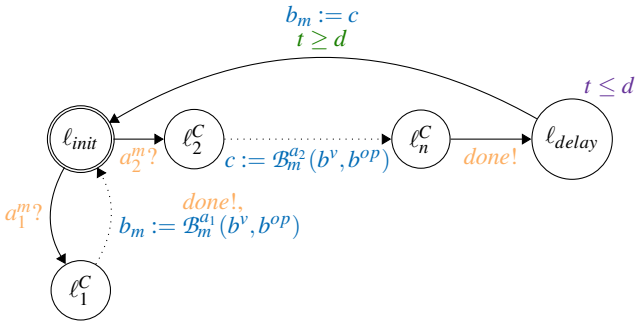


Figure 3: An example of a TA-ACT-R module with an instantaneous actions (a_1^m) returning to ℓ_{init} without time passing, and an action a_2^m delaying for a time d in location ℓ_{a_2}. Dotted edges may be realised by not only one direct edge, but may include complex behaviour or communication with other automata (without time being allowed to pass) e.g. an environmental model.

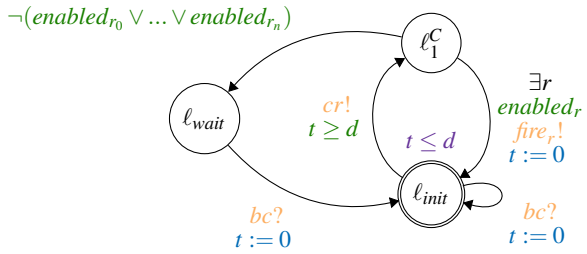


Figure 4: An example of a TA-ACT-R procedural. A rule execution cycle starts in ℓ_{init}, and enters ℓ_1^C after time d synchronising on cr. If there is a rule r with $enabled_r$ true, the rule is fired, if not the procedural enters ℓ_{wait} waiting for bc. The existential quantification over r does function as a short hand for a specialised copy of the edge existing for each production rule.

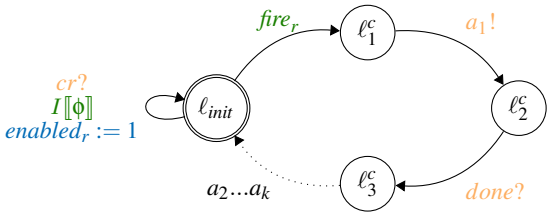


Figure 5: An example of a production rule of TA-ACT-R. The precondition is evaluated on the self loop of ℓ_{init}. Beginning at the $fire_r$ synchronisation, the rule executes actions by synchronising on a_i and *done* (analogous for actions $a_2, \ldots, a_k$).

ACT-R tool, this partition corresponds to the pattern of action, value, and comparator. E.g. the action to remember something, the value blue for a colour slot, and the equality comparator for the colour slot, together formulating the action to remember something blue. The other channel of P-ACT-R that passes arguments of chunk type, $harvest(\gamma)$, is realised in the same way by the combination of a channel (*harvest*) and shared variables on the receiving side.

Functionality of the channels and buffers directly matches the one of P-ACT-R. A module may either make all its computations within no time after receiving the action, or it may start a longer lasting computation and may later finish sending the *bc* message. The computation of the effect of an action is indicated by the update '$c := \mathcal{B}_m^a$' in Figure 3, where the function $\mathcal{B}_m^a$ models the actual computation of a resulting chunk from a and the provided arguments. After completing the computation of effects (without time passing in between), the module synchronises on channel *done* with the rule which issued the command. This cycle of a_n^m, updates, and *done* realises the synchronous execution of actions in the P-ACT-R model. In both cases, timely or delayed completion, the module must be able to accept all action-channels (a_i^m) without blocking while time in the model passes (probably skipping the current computation for a new action, or discarding the action internally).

Module automata never cause deadlocks and we can observe that they do not synchronise on channels of other modules. Neither do they change any variables that belong to other modules (except for *harvest*). Note, that modules are not prohibited from communicating with, e.g., a model of the experimental environment as long as this communication does not interfere with the architecture.

Procedural

Given a set R of rule identities, the TA-ACT-R model of the procedural is the timed automaton $\mathcal{A}_p = (L, B, X, V, I, E, \ell_{ini})$ with channels $B = \{bc, cr, fire(r) \mid r \in R\}$, variables $V = \{enabled_r\}$, and locations and transitions as given by Figure 4. The procedural does not write any variables, but may read the $enabled_r$ variables of rules (modelling E^r, see below). Consequently the procedural of TA-ACT-R does never deadlock, yet it may wait for a *bc* message forever if there is no rule applicable and no module with pending actions.

Rules

The timed automata model of a rule is $\mathcal{A}_r = (L, B, X, V, I, E, \ell_{ini})$ with channels $B = \{fire_r, cr, a_1, ..., a_n\}$ (where $a_1, ..., a_n$ follow the partitioning of actions, slot values and arguments of the modules as discussed above) and variables $V = \{enabled_r\}$. Locations and transitions are given by Figure 5.

Semantically, a TA-ACT-R rule can wait arbitrarily long in its *idle* location (see Figure 5). If the guard is enabled (i.e. the interpretation I of the precondition ϕ is fulfilled), a rule can synchronise on cr, setting $enabled_r$ to true on the edge. If the production system decides to fire production rule r, it synchronises on $fire(r)$, which results in a sequence of edges executing without time advancing. The edges are labelled in sequence with the actions $a_1, ..., a_n$, using b_m^v and b_m^{op} as described in the module section. After each action, the rule waits for the module to finish its initial computations by synchronising on *done*. After the sequence of edges, the rule automaton returns to its *idle* location.

Equivalence of Computations

Assume there is a cognitive model $\mathcal{M}$ consisting of the parallel composition of processes $p, m_1, \ldots, m_N$ (the architecture) and rule processes $c_{r_1}, \ldots, c_{r_M}$. In the following, we want to establish that the behaviour (the set of computation paths) of the corresponding TA-ACT-R model $\mathcal{N}$ is related to the behaviour of the cognitive model in a certain way which in particular allows us to conclude from the reachability of configurations in $\mathcal{N}$ to the reachability in $\mathcal{M}$. Since there exist effective reachability checkers for timed automata (cf. (Langenfeld et al., 2019)) we then obtain an effective procedure to check cognitive models for reachability properties such as absence of deadlocks.

The relation between a P-ACT-R model $\mathcal{M}$ and its TA-ACT-R model $\mathcal{N}$ is a weak bisimulation. That is, there is a so-called **simulation relation** on the configurations c of $\mathcal{M}$ and configurations $\langle \vec{\ell}, \nu \rangle$ of $\mathcal{N}$ such that (a) the initial configurations are related, (b) for each transition sequence of $\mathcal{M}$ from a stable configuration c to the next stable configuration c', if c is related to $\langle \vec{\ell}, \nu \rangle$, then there is a transition sequence in $\mathcal{N}$ to a configuration $\langle \vec{\ell}', \nu' \rangle$ such that c' and $\langle \vec{\ell}', \nu' \rangle$ are related, and (c) $\mathcal{M}$ can simulate transition sequences of $\mathcal{N}$.

A central role in the definition of the simulation relation plays the location ℓ_{wait} of the procedural (the production cycle delay location). A configuration $\langle \vec{\ell}, \nu \rangle$ of $\mathcal{N}$ that has just entered ℓ_{wait} characterises stability in the TA-ACT-R model. Then a stable configuration c of $\mathcal{M}$ and a stable configuration $\langle \vec{\ell}, \nu \rangle$ of $\mathcal{N}$ are related if the state of the buffers of a module m resp. M is equal $c[\![B^m]\!] = \nu(b_M)$ (similarly for Q^m), and for all production rules r it holds that $c[\![E^r]\!] = \nu(enabled_r)$.

If we now assume that the guard expression $I[\![\phi]\!]$ (cf. Figure 5) correctly implements the rule enabledness checks in $\mathcal{M}$, and that the updates $c := \mathcal{B}_m(b^v, b^{op})$ (cf. Figure 3) correctly implement the module behaviour, we are able to show that $\mathcal{N}$ weakly simulates $\mathcal{M}$ by matching the transition sequences in Figure 2 step by step against transitions of the timed automata. Here, the network of timed automata will sometimes need two or more transitions to counter one transition in $\mathcal{M}$, for example with the synchronisation on *done* which is not present in $\mathcal{M}$. The other direction ($\mathcal{M}$ weakly simulates $\mathcal{N}$) follows similarly.

Conclusion

We have presented P-ACT-R, a new abstract, operational, formal model for the cognitive architecture ACT-R. The new model faithfully reflects the structural aspects of cognitive models and the interactions between their components. In contrast to earlier formalisations, P-ACT-R explicitly proposes structural and behavioural interfaces for model components and thereby provides a convenient framework for architectural research. The example of TA-ACT-R shows that the P-ACT-R model can be concretised into fully operational models that can be used with tools for step-wise simulation and exhaustive state space exploration, the (formerly) ad-hoc model TA-ACT-R (Langenfeld et al., 2019) is thereby supplied with a solid formal underpinning.

On the level of TA-ACT-R, we also see how the new approach supports rigorous architecture research. Adding a new module to the TA-ACT-R model amounts to the development of a new timed automaton. To guide this process, the new automaton should realise the desired module behaviour behind the well-defined interface and can then be checked for preserving properties like absence of deadlocks as discussed above. Changing existing module automata for experiments is even easier and guaranteed not to interfere with other components.

For future work, we will explore the relation between the different existing formalisations of the ACT-R framework, such as (Albrecht & Westphal, 2014) as well as continuing to explore the application of P-ACT-R on TA-ACT-R, starting with modelling the defined interfaces within the timed automaton formalisation.

Acknowledgments

This work was supported by the DFG in the project 'Abstract Cognitive Models'.

References

Albrecht, R., & Ragni, M. (2014). Spatial Planning: An ACT-R Model for the Tower of London Task. In C. Freksa, B. Nebel, M. Hegarty, & T. Barkowsky (Eds.), *Spatial Cognition IX* (pp. 222–236). Springer.

Albrecht, R., & Westphal, B. (2014). F-ACT-R: Defining the Architectural Space. *Cogn. Processing*, *15*, 79-81.

Alur, R., & Dill, D. L. (1994). A theory of timed automata. *Theoretical Computer Science*, *126*(2), 183–235.

Anderson, J. R. (2007). *How Can the Human Mind Occur in the Physical Universe?* Oxford University Press.

Bothell, D. (2013). *ACT-R Tutorial.* Retrieved from http://act-r.psy.cmu.edu/actr6/.

Gall, D., & Frühwirth, T. (2018). An operational semantics for the cognitive architecture ACT-R and its translation to constraint handling rules. *ACM TCL*, *19*(3), 22:1–22:42.

Gall, D., & Frühwirth, T. W. (2014). A formal semantics for the cognitive architecture ACT-R. In *LOPSTR* (Vol. 8981, pp. 74–91). Springer.

Langenfeld, V., Westphal, B., Albrecht, R., & Podelski, A. (2018). But does it really do that? Using formal analysis to ensure desirable ACT-R model behaviour. In *CogSci 2018* (pp. 659–664).

Langenfeld, V., Westphal, B., & Podelski, A. (2019). On formal verification of ACT-R architectures and models. In *CogSci 2019* (pp. 618–624).

Ragni, M., Sauerwald, K., Bock, T., Kern-Isberner, G., Friemann, P., & Beierle, C. (2018). Towards a formal foundation of cognitive architectures. In *CogSci 2018* (pp. 2321–2326).

Schultheis, H. (2009). Computational and Explanatory Power of Cognitive Architectures: The Case of ACT-R. In *Proc. 9th Int. Conf. on Cognitive Modeling* (pp. 384–389).

Stewart, T. C., & West, R. L. (2007). Deconstructing and Reconstructing ACT-R: Exploring the Architectural Space. *Cog. Sys. Research*, *8*(3), 227–236.