A Theory of Service Dependency

Mats Neovius Luigia Petre Kaisa Sere
Åbo Akademi University, Faculty of Science and Engineering
Turku Centre for Computer Science (TUCS)
Turku, Finland

Service composition has become commonplace nowadays, in large part due to the increased complexity of software and supporting networks. Composition can be of many types, for instance sequential, prioritising, non-deterministic. However, a fundamental feature of the services to be composed consists in their dependencies with respect to each other. In this paper we propose a theory of service dependency, modelled around a dependency operator in the Action Systems formalism. We analyze its properties, composition behaviour, and refinement conditions with accompanying examples.

1 Introduction

Dependency can be of several types, for instance we can think of type and format dependencies between data producers and data consumers or of signature and semantic dependencies between service providers and service users. Moreover, when getting serviced by various service providers, we depend on them in ways not yet formally understood. As our contemporary digital activities (such as online banking or shopping) are based on service providers, that use in their turn other service providers, we need to better understand the composition types between all the involved services. More interestingly, composing services that depend on each other in various ways adds a special flavor to the problem.

In this paper we define dependency via a specific operator and analyze its properties especially in correlation with previously defined and studied composition operators. Our study is developed in the Action Systems [6] formal framework. Analysis includes examining basic algebraic properties in the formal Action Systems framework as well as detailing how refinement applies to dependency.

Action Systems is a state-based formal method for modeling distributed systems. Introduced in 1983, when CSP [17] and CCS [21] where the major modeling formalisms, it differed from them in that it proposed an overall approach of a system. CSP and CCS are process algebras, modeling the behavior of the processes of a system, together with their interactions. The basic idea of Action Systems is to model the overall system behavior, often in an abstract manner. The genericity of such abstractions are not problematic because Action Systems is built around the concept of *refinement*: a specification can be correctly developed from a more abstract to a more concrete form, by following formal rules for such developments. Nowadays, Action Systems are very resemblant of the Event-B [1] formal method, which is, in fact, based on it and on the B-method [2]. Notable for Event-B is an associated theorem prover, the Rodin platform [3], in which one can edit the system models and get automatically the proof obligations to prove, in order for the models to be correct with respect to various properties. Action Systems remains to this day much more general and flexible than Event-B, even if it has the downside of missing an equivallent tool platform. However, we set up our study of dependency in Action Systems, because of its flexibility. Once we understand all the concepts well, we plan to move our understanding into the Rodin platform, in form of a theory of actions. This is still very preliminary; we mention some thoughts on this in the conclusions.

Hence, the contribution of this paper consists in modelling dependency in a state-based formalism via a dedicated operator. We analyse fundamental properties of this operator, including refinement, and

Eerke Boiten, John Derrick & Steve Reeves (Eds.):
Refinement Workshop 2015 (Refine '15)
EPTCS 209, 2016, pp. 112–128, doi:10.4204/EPTCS.209.9

© M. Neovius, L. Petre & K. Sere
This work is licensed under the
Creative Commons Attribution License.

emphasise various examples relevant to our discussion. We believe this is the beginning of a series of studies on dependency, as it has become such an intricate phenomenon in our digitalised society.

We proceed as follows. In Section 2 we discuss Action Systems to the extent needed in this paper, after which in Section 3 we introduce and analyse the dependency operator. Refinement laws for dependency are studied in Section 4, and in Section 5 we outline an example that illustrates some of the introduced concepts. We conclude the paper with highlights of future work in Section 6.

2 Action Systems - a Revisit

The Action System framework was introduced by Back and Kurki-Suonio in 1983 [6] for modeling distributed systems. It has been investigated and extensively developed for about two decades, prominently by Back, Sere, von Wright, Sekerinski, Butler, and colleagues [8, 9, 10, 11, 15, 25, 26, 27, 28, 30]. An Action Systems overview appears in [23].

In the following we revisit some of the fundamental building blocks of Action Systems that will then be employed for studying dependency.

2.1 Preliminaries

An *action system* consists of a *state* that can be evaluated and modified by a finite set of *actions*. The state models the problem domain of the system via a finite set of *variables*: at any moment, the state contains the values of these variables. The state can also be described as a predicate understood as the conjunction of predicates describing the values of the variables, for instance the state can be described as $x = 5 \wedge y = 10$, where x and y are the state variables. The value of a variable can be read and modified by an action. Each action can read and modify a subset of the state variables. An action system is not necessarily regarded in isolation, but as a part of a more complex system. The rest of the system (the *environment*) communicates with the action system using different mechanisms such as global variables or exported procedures [28].

An action system $\mathscr{A}$ has the following form:

$$\mathscr{A} \quad = \quad |[\mathsf{var}\ x \bullet S_0\ ;\ \mathsf{do}\ A_1\ [\!]\ \ldots\ [\!]\ A_m\ \mathsf{od}\ ]|\ y \tag{1}$$

Here $x = x_1, \ldots, x_n$ are the *variables* of the system $\mathscr{A}$, S_0 is a statement initializing them, while A_i, $i = 1, \ldots, m$, are the *actions* of the system. Variables in x may be *exported*, in the sense that they can be read, or written, or both read and written by environment actions. We refer to a local variable of $\mathscr{A}$ that is not exported as *private*. The *imported variables* $y = y_1, \ldots, y_k$ are declared in the environment of $\mathscr{A}$. We assume that $x \cap y = \emptyset$ and refer to $x \cup y$ as the *global* variables of $\mathscr{A}$.

Notation-wise, we observe that the boundary of the system $\mathscr{A}$ is denoted with brackets $|[...]|$; the entities inside the brackets are defined within $\mathscr{A}$ while the entities outside the brackets (namely, y) are not. Inside $\mathscr{A}$ we observe the sequence ; with its two parts: the first consists of the variable declaration and initialisation and the second consists of a do ... od loop containing the actions separated by the non-deterministic choice operator $[\!]$. We explain the execution model shortly, upon understanding the form and semantics of actions.

An $action$ is an atomic statement that can change the values of the local or global variables of the action system. An action A can be described by the following grammar:

$$A \quad ::= \quad abort \mid skip \mid x :\!= v \mid p \rightarrow A \mid A\ [\!]\ A. \tag{2}$$

Here x is a list of variables, v a list of values, and p a predicate on the state variables. Intuitively, '$abort$' is the action that always deadlocks, '$skip$' is the stuttering action, '$x := v$' is a multiple assignment, '$p \rightarrow A$' is a guarded action, executable only when p holds, and '$A_1 \, [] \, A_2$' is the nondeterministic choice among actions 'A_1' and 'A_2'. We note here that the actions (2) are suitable for specification, being rather abstract. For instance, the more deterministic sequential and prioritising compositions are missing, although they are well known for Action Systems. We discuss them shortly.

The semantics of an action A is described in terms of the weakest precondition predicate transformer wp, in the style of Dijkstra [14]. The weakest precondition predicate transformer relates the state of the system after an action A has taken place (the postcondition q of A) to the widest possible state of the system before the action A has taken place (weakest precondition of A with respect to q). In this way, it completely describes an action by defining from what precondition one should start in order to arrive at a desired postcondition. Given a postcondition q, the function $wp(A,q)$ is defined below for actions (2):

$$\begin{array}{lcl} wp(abort, q) & = & false \\ wp(skip, q) & = & q \\ wp(x := v, q) & = & q[x := v] \\ wp(p \rightarrow A, q) & = & (p \Rightarrow wp(A, q)) \\ wp(A_1 \, [] \, A_2, q) & = & wp(A_1, q) \wedge wp(A_2, q). \end{array}$$

The details of the definition of this function are studied elsewhere [8, 28]. Here we just discuss an intuitive understanding of this semantical way of defining actions. Consider action $skip$: it is clear that, to arrive at a postcondition q by doing nothing in terms of state changes, one should start from the same precondition q. Consider also the assignment $x := v$: what we want to happen when such an assignment is executed is that the variables x should end up with values v and all the other variables should keep their values; hence, to arrive at a postcondition q after executing $x := v$, we need to replace all occurrences of x in q with v. The action $abort$ is a special case, denoting an abandonment of computation; to arrive at postcondition q when such an abandonment takes place is impossible, hence, there is no state from where one could get to q via $abort$; hence, $wp(abort,q) = false$. For action $p \rightarrow A$, we execute A to get to q, but only when p holds; hence, we need to start in a state where $wp(A,q)$ holds when p holds. When p does not hold, then nothing happens, so one can start from anything; in this case 'anything' is modelled by $true$. The nondeterministic choice $A_1 \, [] \, A_2$ is perhaps the most interesting of the actions (2), as it means that either A_1 or A_2 can take place, but there is no way to know in advance which of them actually takes place; hence, in order to get to q with $A_1 \, [] \, A_2$ we must be prepared and start from a state in which both $wp(A_1,q)$ and $wp(A_2,q)$ hold.

A useful property of the wp predicate transformer is to characterize the termination of an action: if $wp(A,true) = true$, then we say that A terminates (also referred to as *always terminating* [28]). This can be interpreted so that $true$ is a postcondition describing the state of the system without any restriction: the variables could have any values. Thus, if for an action A we have that $wp(A,true) = true$, then we know nothing about how this action works except that it terminates.

The predicate transformer wp is also conjunctive, as defined below; this property is useful when doing wp-calculations, as we will see in Section 3.

$$wp(A, p \wedge q) = wp(A, p) \wedge wp(A, q) \tag{3}$$

Equality of actions Based on the wp predicate transformer we can compare various (compositions of) actions. We are interested only in the input-output behaviour of actions, in terms of state changes, hence we consider two actions to be *equal* if they always start from the same weakest precondition in order to achieve the same postcondition, for all possible postconditions:

$$A_1 = A_2 \quad \text{iff} \quad \text{for all } q: \quad wp(A_1, q) = wp(A_2, q) \tag{4}$$

Enabledness An important property of an action is its *enabledness*, defined via the action's *guard*: we say that an action is *enabled* when its guard holds. We are interested in 'functional' states when modeling, namely those from where actions achieve useful postconditions; for this, we exclude those states from which an action establishes postcondition $false$, which models an impossible state. Hence, we define the guard of A, denoted $g(A)$, as $\neg wp(A, false)$: this gives those states in which action A behaves in a functional way. The actions (2) have the following guards:

$$\begin{array}{lcl} g(abort) & = & true \\ g(skip) & = & true \\ g(x := v) & = & true \\ g(p \to A) & = & p \wedge g(A) \\ g(A_1 \,[\!]\, A_2) & = & g(A_1) \vee g(A_2) \end{array} \tag{5}$$

Actions whose guards are always true are called *always enabled* [28], for instance an assignment or a $skip$ action are always enabled. The action $p \to A$ is enabled when both p and $g(A)$ hold: $\neg wp(p \to A, false) = \neg(p \Rightarrow wp(A, false)) = \neg(\neg p \vee wp(A, false)) = p \wedge \neg wp(A, false) = p \wedge g(A)$. A similar calculation leads to the formula (5) for $g(A_1 \,[\!]\, A_2)$.

When we think of an action A having the guard $g(A)$, the guardless 'rest' of the action is syntactically referred to as the *body* $b(A)$ of action A, so that $g(b(A)) = true$. Thus, we can write action A as $A = g(A) \to b(A)$. The study of action guards appears in the Action Systems literature, for instance in [8, 26, 28], to support various other constructs. As enabledness is very important for service dependency, in this paper we study guards themselves in more detail. Notation-wise, whenever convenient we write gA instead of $g(A)$ and similarly we write bA instead of $b(A)$.

Example 1 Lets assume we have a simple road crossing as the one illustrated in Figure 1. We model the two crossing roads as four segments labelled A, B, C, D. The action system $\mathscr{C}\mathscr{C}_B$ below describes a simple behavior of a car entering the crossing at segment B.

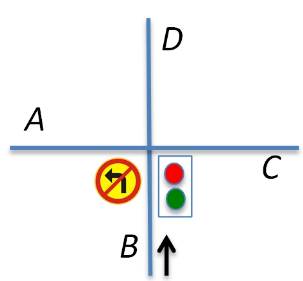

Figure 1: A simple crossing with one car

$$
\begin{aligned}
\mathscr{C}\mathscr{C}_B \quad = \quad & |[\mathsf{var}\ light : \{green,\ red\}, loc : \{A,\ B,\ C,\ D\} \bullet \\
& light,\ loc :\!= red,\ B\ ; \\
& \mathsf{do} \\
& \ (A_1)\ light = red \rightarrow light :\!= green \\
& [\!]\,(A_2)\ light = green \rightarrow light :\!= red \\
& [\!]\,(A_3)\ loc = B \wedge light = green \rightarrow loc :\!= C\ [\!]\ loc :\!= D \\
& \mathsf{od}\ ]|
\end{aligned}
$$

The car on segment B can only go through the crossing if the light is $green$. The state of $\mathscr{C}\mathscr{C}_B$ is described by two variables $light$ and loc, the first modeling the crossing lights and the second the location of the car on one of the four segments A, B, C, D. We can see examples of assignments in this small Action System, as well as of guards and non-determinism. Actions A_1 and A_2 switch between the lights. Action A_3 models that the location of the car can change from B to C or D only when the car is at location B ($loc = B$) and the crossing lights are green ($light = green$). In this case, the location is non-deterministically changed to either C or D: $loc :\!= C\ [\!]\ loc :\!= D$.

2.2 The execution model

The execution of an action system $\mathscr{A}$ as in (1) is the following. The initialisation S_0 sets the variables to some specific values. Then, from the enabled actions, one is non-deterministically chosen and executed: this means that the chosen action changes the values of its accessed variables in a way that is determined by the action body. The variables that are not accessed by that action keep their values unchanged. The execution of any action is atomic: this means that, once the action is selected for execution, it will execute without interference from other actions. The computation terminates when no action is enabled. This means that the state will evolve no more, fixing the final values of the variables forever. The action system $\mathscr{C}\mathscr{C}_B$ is non-terminating, as the lights will keep switching via actions A_1 and A_2. Upon initialisation, after the lights become $green$, both actions A_2 and A_3 are enabled: if A_2 is chosen for execution, then the lights change back to red, and then only A_1 is enabled. When $lights = green$ and A_3 is chosen for execution, then the car will go straight on, advancing to segment D or to the right, advancing to segment C.

Such an execution model is similar to Dijkstra's guarded iteration statement [14], showing Action Systems can model *sequential executions*. Parallelism can also be modelled in the framework, by *interleaving*. In such a *parallel execution* model, actions that do not access each other's variables and are enabled at the same time can be executed in parallel. This is possible because their sequential execution in any order has the same result. Execution models are detailed in [4, 5, 10].

Execution of any action cannot be guaranteed in the Action System framework. This is due to assuming *no notion of fairness* in the model [4]. Fairness [20], as a property that concurrent systems may have, can be of several forms. One of the most used forms, referred also as *strong* fairness, means that an action is infinitely often executed if it is infinitely often enabled. Having no assumptions of fairness implies that true non-determinism can be modeled with Action Systems. Also, properties proved for a sequential execution of an action system $\mathscr{A}$ as in (1) still hold when a parallel execution is assumed for $\mathscr{A}$ [4, 7]. This feature is important because the theory supporting proofs about sequential executions is rich, see for example [14, 16].

2.3 More deterministic composition operators

The actions (2) can model abstract specifications, that include non-determinism as an abstraction mechanism. We now focus on two action composition operators that enable more determinism in the specifications. We extend the grammar (2) as follows:

$$A \quad ::= \quad \ldots \mid A \mathbin{/\!/} A \mid A \,;\, A. \tag{6}$$

Here '$A_1 \mathbin{/\!/} A_2$' is the prioritising composition of two actions 'A_1' and 'A_2' and '$A_1 ; A_2$' is the sequential composition of two actions 'A_1' and 'A_2':

$$\begin{array}{lcl} A_1 \mathbin{/\!/} A_2 & = & A_1 \, [\!] \, \neg gA_1 \rightarrow A_2 \\ A_1 \,;\, A_2 & = & gA_1 \wedge wp(bA_1, gA_2) \rightarrow bA_1 \,;\, bA_2 \end{array} \tag{7}$$

Prioritising composition One reason behind defining the $/\!/$ operator between actions if that of *coordination*. The underlying execution model of Action Systems is non-deterministic, i.e., the scheduling of certain actions for execution cannot be guaranteed. However, when modeling coordination we need to enforce the execution of specific actions. The notion of coordination was therefore defined in terms of prioritising composition for Action Systems in [15, 27]. We say that the action A_1 *coordinates* the action A_2. Essentially, action A_1 has a higher priority than action A_2: A_1 can be executed if it is enabled, while A_2 can be executed if it is enabled *and* A_1 is not enabled.

Example 2 To see an example of prioritising composition, lets think again about a simple crossing, but this time without the crossing lights and with two cars trying to pass through it, as illustrated in Figure 2. Assume both cars just want to continue on their roads and so, without crossing lights, we need to take into account the right-of-way priority: the car coming from the East will have priority over the car coming from the South. We have the following two actions modeling the desired movement of the two cars:

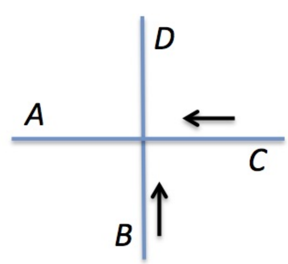

Figure 2: A simple crossing with two cars

$$\begin{array}{ll} (A_4) & loc_1 = B \rightarrow loc_1 :\!= D \\ (A_5) & loc_2 = C \rightarrow loc_2 :\!= A \end{array}$$

Here, action A_4 models the desired movement of the car from the South, while action A_5 models the desired movement of the car from the East. Assuming right-hand traffic, the right-of-way is modeled by the prioritised composition $A_5 \mathbin{/\!/} A_4$: action A_4 will execute when enabled and when action A_5 is disabled.

The weakest precondition with respect to a predicate q and the guard of $A_1 \mathbin{/\!/} A_2$ are, respectively:

$$\begin{array}{lcl} wp(A_1 \mathbin{/\!/} A_2, q) & = & wp(A_1, q) \wedge (gA_1 \vee wp(A_2, q)) \\ g(A_1 \mathbin{/\!/} A_2) & = & gA_1 \vee gA_2 \end{array} \tag{8}$$

Sequential composition For specifying sequentiality, we use the ; operator. This is, in fact, one of the fundamental operators in [14], defined as follows: $A_1 \,;\, A_2$ behaves as A_1 if A_1 is enabled, then, when A_1 terminates, as A_2 if A_2 is enabled; otherwise, the sequence $A_1 \,;\, A_2$ is not enabled [11]. The weakest precondition with respect to a predicate q and the guard of $A_1 \,;\, A_2$ are, respectively:

$$\begin{array}{lcl} wp(A_1 \,;\, A_2, q) & = & wp(A_1, wp(A_2, q)) \\ g(A_1 \,;\, A_2) & = & gA_1 \wedge wp(bA_1, gA_2) \end{array} \tag{9}$$

A useful construct for working with actions is also the *assumption* $[p]$, where p is a predicate. We have that $wp([p], q) = (p \Rightarrow q)$ and that $g([p]) = p$. The meaning of $[p]$ is that is behaves as $skip$ if p holds and as $abort$ otherwise. Its usefulness comes from the fact that an action $p \rightarrow A$ is defined as $[p] \,;\, A$:

$$p \rightarrow A \quad = \quad [p] \,;\, A \tag{10}$$

Based on assumption and sequential composition, we define the following:

$$A_1 \text{ enables } A_2 \quad = \quad wp(A_1, gA_2) \tag{11}$$

$$A_1 \text{ cannot disable } A_2 \quad = \quad gA_2 \Rightarrow wp(A_1, gA_2) \tag{12}$$

$$A_1 \text{ cannot enable } A_2 \quad = \quad \neg gA_2 \Rightarrow wp(A_1, \neg gA_2) \tag{13}$$

Definition (11) essentially means that action A_1 terminates and establishes as postcondition gA_2: if A_2 was enabled before A_1 took place, then A_1 did not disabled it and if A_2 was disabled before A_1 took place, then A_1, via its state changes, enabled A_2. Understanding this feature is important for the dependency operator that we define in the next section. Essentially, A_1 enables A_2 means that, if A_1 is enabled, then it will execute and enable A_2. This is also observable from the following calculation: $wp(A_1, gA_2) = wp([gA_1] \,;\, bA_1, gA_2) = wp([gA_1], wp(bA_1, gA_2)) = (gA_1 \Rightarrow wp(bA_1, gA_2))$. Definition (12) is a stronger condition than (11), modelling that A_2 was enabled before A_1 took place and A_1 did not disabled it. More detailed calculations reduce definition (12) to $gA_1 \wedge gA_2 \Rightarrow wp(bA_1, gA_2)$. Definition (13) models that, if A_2 was disabled before A_1 took place, then A_1 did not enabled it. More detailed calculations reduce definition (13) to $gA_1 \wedge \neg gA_2 \Rightarrow wp(bA_1, \neg gA_2)$.

In the context of the above definitions, we rephrase the guard $g(A_1 \,;\, A_2)$ of $A_1 \,;\, A_2$ (9) as follows: A_1 should be enabled and should enable A_2 upon its termination. We observe that A_2 does not need to be enabled before A_1 terminates.

We present below two more properties of the assumption construct, that are instrumental in this paper. First, we can split an assumption made of a conjunction of predicates into sequential (and commutative) assumptions, as described in (14):

$$
\begin{aligned}
& wp([a \wedge b], q) \\
= & \{\text{weakest precondition of assumption}\} \\
& a \wedge b \Rightarrow q \\
= & \{\text{logic}\} \\
& \neg(a \wedge b) \vee q \\
= & \{\text{logic}\} \\
& (\neg a \vee \neg b) \vee q \\
= & \{\vee \text{ is associative}\} \\
& \neg a \vee (\neg b \vee q) \\
= & \{\text{logic}\} \\
& a \Rightarrow (b \Rightarrow q) \\
= & \{\text{weakest precondition of assumption, twice}\} \\
& wp([a], wp([b], q)) \\
= & \{\text{weakest precondition of } ; \text{ twice}\} \\
& wp([a] ; [b], q))
\end{aligned}
\tag{14}
$$

Property (14) can be thus written as $[a \wedge b] = [a] ; [b]$ and this can also be written as $[a \wedge b] = [b] ; [a]$, as conjunction and disjunction are commutative operators.

Second, we need to discuss what assumption $[wp(A, g)]$ means in a sequential composition. First, assume $wp(A, g)$ holds. This is then equivallent to $[true] ; A ; [g]$, meaning that, when A is executed, it establishes postcondition g. Assume $wp(A, g)$ holds in the following sequential composition: $[wp(A, g)] ; X ; A ; Y$. We can then rewrite $[wp(A, g)] ; X ; A ; Y$ as $X ; A ; [g] ; Y$. We formalize this in property (15):

$$[wp(A, g)] ; X ; A ; Y \quad = \quad X ; A ; [g] ; Y \tag{15}$$

3 The Dependency Operator

Having briefly reviewed the composition operators of actions, we now turn our attention to modelling dependency. The *dependency* operator, $\backslash\backslash$, has already been introduced [22] as follows:

$$A \quad ::= \quad \ldots \mid A \backslash\backslash A, \tag{16}$$

where

$$A_1 \backslash\backslash A_2 \quad = \quad gA_1 \wedge gA_2 \rightarrow A_1 ; A_2 \tag{17}$$

The weakest precondition with respect to a predicate q and the guard of $A_1 \backslash\backslash A_2$ are, respectively:

$$
\begin{aligned}
wp(A_1 \backslash\backslash A_2, q) & = gA_1 \wedge gA_2 \Rightarrow wp(A_1, wp(A_2, q)) \\
g(A_1 \backslash\backslash A_2) & = gA_1 \wedge gA_2 \wedge wp(bA_1, gA_2)
\end{aligned}
\tag{18}
$$

The interpretation of this operator is as follows. We say that A_1 depends on A_2, because it needs A_2 to be enabled before and after its own (A_1's) execution. In its turn, in the simple sequential composition $A_1 ; A_2$ it is only necessary for A_2 to be enabled after A_1's execution. To better see this difference between the two action compositions, we decompose $A_1 ; A_2$ and $A_1 \backslash\backslash A_2$, based on (10), as follows:

$$A_1 ; A_2 \quad = \quad [gA_1] ; bA_1 ; [gA_2] ; bA_2 \tag{19}$$

$$A_1 \backslash\backslash A_2 \quad = \quad [gA_1] ; [gA_2] ; bA_1 ; [gA_2] ; bA_2 \tag{20}$$

Hence, gA_2 acts as an *invariant* for bA_1, as observed in 20: it should hold before and after bA_1 takes place.

Example 3 Let us assume the situation of a customer waiting to be served at a bank. In Finland, customers get each a queue number from a machine and wait for their number to be displayed on a screen, with an indication to which cashier to proceed. Once in front of the right cashiers, customers get serviced and get each a receipt for their service, printed by a printer at the cashier's desk. Assume we have three actions for a customer-server provider pair:

- Action $A_1 = gA_1 \to bA_1$, where gA_1 models that a customer has a queue number and bA_1 models that the customer waits to be served;
- Action $A_2 = gA_2 \to bA_3$, where gA_2 models that the customer's number is called by a cashier (displayed on a screen) and bA_2 models that the cashier provides the desired service as well as commands the printing of the receipt;
- Action $A_3 = gA_3 \to bA_3$, where gA_3 models that the (service provider's) printer has paper and bA_3 models that this printer prints the receipt.

We have obviously a sequence between A_1 and A_2: $A_1 \,;\, A_2 = [gA_1] \,;\, bA_1 \,;\, [gA_2] \,;\, bA_2$. The condition gA_2 (of the customer's number being called) does not need to hold before bA_1 takes place: it needs to hold after bA_1 took place. However, there is a dependency between A_2 and A_3: $A_2 \backslash\backslash A_3 = [gA_2] \,;\, [gA_3] \,;\, bA_2 \,;\, [gA_3] \,;\, bA_3$. The condition gA_3 that the printer has paper needs to hold before the cashier presses the 'print' button on his screen; if gA_3 does not hold before that, then the cashier needs to do some other actions, for instance replenishing the paper.

We will now study several properties of the dependency operator. We begin by observing the following:

$$
\begin{aligned}
& A_1 \backslash\backslash A_2 \\
= & \{\text{definition (17) of } \backslash\backslash\} \\
& gA_1 \wedge gA_2 \to A_1 \,;\, A_2 \\
= & \{\text{definition (7) of } ;\} \\
& gA_1 \wedge gA_2 \to gA_1 \wedge wp(bA_1, gA_2) \to bA_1 \,;\, bA_2 \\
= & \{\text{assumption definition (10)}\} \\
& [gA_1 \wedge gA_2] \,;\, [gA_1 \wedge wp(bA_1, gA_2)] \,;\, bA_1 \,;\, bA_2 \\
= & \{\text{assumption properties, ';' is associative, logic}\} \\
& [gA_1 \wedge gA_2 \wedge wp(bA_1, gA_2)] \,;\, bA_1 \,;\, bA_2 \\
= & \{\text{assumption definition (10)}\} \\
& gA_1 \wedge gA_2 \wedge wp(bA_1, gA_2) \to bA_1 \,;\, bA_2
\end{aligned}
\tag{21}
$$

Hence, we can use either (17), (20) or (21) to express $A_1 \backslash\backslash A_2$.

Commutativity and Associativity The dependency operator $\backslash\backslash$ is based on the sequential composition ;. We know from basic theory that $A_1 \,;\, A_2 = A_2 \,;\, A_1$ only in special cases. In general, ; is not commutative and correspondingly, $\backslash\backslash$ is not commutative either.

We now consider associativity of $\backslash\backslash$. We have the following:

$$
\begin{aligned}
& (A_1 \backslash\backslash A_2) \backslash\backslash A_3 \\
= & \{(20)\} \\
& ([gA_1] ; [gA_2] ; bA_1 ; [gA_2] ; bA_2) \backslash\backslash A_3 \\
= & \{(20), (21), (14)\} \\
& [gA_1] ; [gA_2] ; [wp(bA_1, gA_2)] ; [gA_3] ; [wp(bA_1 ; bA_2, gA_3)] ; bA_1 ; bA_2 ; bA_3 \\
= & \{(15)\} \\
& [gA_1] ; [gA_2] ; [gA_3] ; bA_1 ; [gA_2] ; bA_2 ; [gA_3] ; bA_3
\end{aligned}
$$

By a similar computation, we have:

$$
\begin{aligned}
& A_1 \backslash\backslash (A_2 \backslash\backslash A_3) \\
= & \{(20), (21), (14), (15)\} \\
& [gA_1] ; [gA_2] ; [gA_3] ; bA_1 ; [gA_2] ; [\mathbf{gA_3}] ; bA_2 ; [gA_3] ; bA_3
\end{aligned}
$$

Since ; is associative, we obtain that $\backslash\backslash$ is associative only when gA_3 is an invariant for A_1, i.e., when A_1 cannot disable A_3 (12):

$$
(\mathbf{A_1} \backslash\backslash \mathbf{A_2}) \backslash\backslash \mathbf{A_3} = \mathbf{A_1} \backslash\backslash (\mathbf{A_2} \backslash\backslash \mathbf{A_3}) \quad \text{iff} \quad \mathbf{gA_3} \Rightarrow \mathbf{wp}(\mathbf{A_1}, \mathbf{gA_3}) \tag{22}
$$

Distributivity We now check the distributivity of $\backslash\backslash$ over ;. We have the following:

$$
\begin{aligned}
& A_1 \backslash\backslash (A_2 ; A_3) \\
= & \{(19)\} \\
& A_1 \backslash\backslash ([gA_2] ; bA_2 ; [gA_3] ; bA_3) \\
= & \{(20)\} \\
& [gA_1] ; [gA_2] ; bA_1 ; [gA_2] ; ([gA_2] ; bA_2 ; [gA_3] ; bA_3) \\
= & \{\text{`;' is associative, logic}\} \\
& ([gA_1] ; [gA_2] ; bA_1 ; [gA_2] ; bA_2) ; [gA_3] ; bA_3 \\
= & \{(19), (20)\} \\
& (A_1 \backslash\backslash A_2) ; A_3
\end{aligned}
$$

Hence, we have that:

$$
\mathbf{A_1} \backslash\backslash (\mathbf{A_2} ; \mathbf{A_3}) = (\mathbf{A_1} \backslash\backslash \mathbf{A_2}) ; \mathbf{A_3} \tag{23}
$$

With respect to the distributivity of ; over $\backslash\backslash$, we have the following calculations:

$$
\begin{aligned}
& A_1 ; (A_2 \backslash\backslash A_3) \\
= & \{(20), (10)\} \\
& [gA_1] ; bA_1 ; ([gA_2] ; [gA_3] ; bA_2 ; [gA_3] ; bA_3) \\
= & \{\text{`;' is associative}, (14)\} \\
& [gA_1] ; \mathbf{bA_1} ; [\mathbf{gA_3}] ; [gA_2] ; bA_2 ; [gA_3] ; bA_3
\end{aligned}
$$

$$
\begin{aligned}
& (A_1 ; A_2) \backslash\backslash A_3 \\
= & \{(19), (10)\} \\
& ([gA_1] ; bA_1 ; [gA_2] ; bA_2) \backslash\backslash ([gA_3] ; bA_3) \\
= & \{\text{`;' is associative}, (20), (15)\} \\
& [gA_1] ; [\mathbf{gA_3}] ; \mathbf{bA_1} ; [gA_2] ; bA_2 ; [gA_3] ; bA_3
\end{aligned}
$$

Hence, the associativity of ; over \\ holds only if gA_3 and bA_1 commute, for instance when $gA_3 = true$ or when bA_1 and gA_3 have no variables in common. We thus have:

$$\mathbf{A_1} \,;\, (\mathbf{A_2} \backslash\backslash \mathbf{A_3}) = (\mathbf{A_1} \,;\, \mathbf{A_2}) \backslash\backslash \mathbf{A_3} \quad \text{iff} \quad [\mathbf{gA_3}] \,;\, \mathbf{bA_1} = \mathbf{bA_1} \,;\, [\mathbf{gA_3}] \tag{24}$$

We now check the distributivity of \\ over [] and // . We have the following:

$$\begin{aligned}
& A_1 \backslash\backslash (A_2 \,[\!]\, A_3) \\
= & \{(10), (5)\} \\
& ([gA_1] \,;\, bA_1) \backslash\backslash ([gA_2 \vee gA_3] \,;\, (A_2 \,[\!]\, A_3)) \\
= & \{(20)\} \\
& [gA_1] \,;\, [gA_2 \vee gA_3] \,;\, bA_1 \,;\, [gA_2 \vee gA_3] \,;\, (A_2 \,[\!]\, A_3)
\end{aligned}$$

$$\begin{aligned}
& (A_1 \backslash\backslash A_2) \,[\!]\, (A_1 \backslash\backslash A_3) \\
= & \{(20)\} \\
& ([gA_1] \,;\, [gA_2] \,;\, bA_1 \,;\, [gA_2] \,;\, bA_2) \,[\!]\, ([gA_1] \,;\, [gA_3] \,;\, bA_1 \,;\, [gA_3] \,;\, bA_3) \\
= & \{(10), \text{`;' is associative}, (14)\} \\
& (gA_1 \wedge gA_2 \rightarrow bA_1 \,;\, A_2) \,[\!]\, (gA_1 \wedge gA_3 \rightarrow bA_1 \,;\, A_3) \\
= & \{\text{properties of} \rightarrow\} \\
& gA_1 \rightarrow ((gA_2 \rightarrow bA_1 \,;\, A_2) \,[\!]\, (gA_3 \rightarrow bA_1 \,;\, A_3)) \\
= & \{(10), (14)\} \\
& [gA_1] \,;\, [gA_2 \vee gA_3] \,;\, bA_1 \,;\, [gA_2 \vee gA_3] \,;\, (A_2 \,[\!]\, A_3)
\end{aligned}$$

Hence, \\ distributes over [] to the left:

$$\mathbf{A_1} \backslash\backslash (\mathbf{A_2} \,[\!]\, \mathbf{A_3}) = (\mathbf{A_1} \backslash\backslash \mathbf{A_2}) \,[\!]\, (\mathbf{A_1} \backslash\backslash \mathbf{A_3}) \tag{25}$$

By a similar proof we can show that \\ distributes over [] to the right as well. We do not show here the proof, for space purposes. Hence, we also have:

$$(\mathbf{A_1} \,[\!]\, \mathbf{A_2}) \backslash\backslash \mathbf{A_3} = (\mathbf{A_1} \backslash\backslash \mathbf{A_3}) \,[\!]\, (\mathbf{A_2} \backslash\backslash \mathbf{A_3}) \tag{26}$$

For checking the distributivity of \\ over // , we have the following calculations:

$$\begin{aligned}
& A_1 \backslash\backslash (A_2 \,/\!/\, A_3) \\
= & \{(7)\} \\
& A_1 \backslash\backslash (A_2 \,[\!]\, \neg gA_2 \rightarrow A_3) \\
= & \{(25)\} \\
& (A_1 \backslash\backslash A_2) \,[\!]\, (A_1 \backslash\backslash (\neg gA_2 \rightarrow A_3)) \\
= & \{(17)\} \\
& (gA_2 \rightarrow A_1 \backslash\backslash A_2) \,[\!]\, (\neg gA_2 \rightarrow A_1 \backslash\backslash \neg gA_2 \rightarrow A_3) \\
= & \{(7)\} \\
& (A_1 \backslash\backslash A_2) \,/\!/\, (A_1 \backslash\backslash \neg gA_2 \rightarrow A_3)
\end{aligned}$$

Hence, \\ distributes over // to the left conditionally, if A_1 cannot enable A_2 (13):

$$\mathbf{A_1} \backslash\backslash (\mathbf{A_2} \,/\!/\, \mathbf{A_3}) = (\mathbf{A_1} \backslash\backslash \mathbf{A_2}) \,/\!/\, (\mathbf{A_1} \backslash\backslash \mathbf{A_3}) \quad \text{iff} \quad \neg \mathbf{gA_2} \Rightarrow \mathbf{wp}(\mathbf{A_1}, \neg \mathbf{gA_2}) \tag{27}$$

For distribution to the right we have:

$$
\begin{aligned}
& (A_1 \mathbin{/\!/} A_2)\backslash\backslash A_3 \\
= & \{(7)\} \\
& (A_1 \,[\!]\, \neg gA_1 \rightarrow A_2)\backslash\backslash A_3 \\
= & \{(26)\} \\
& (A_1\backslash\backslash A_3) \,[\!]\, ((\neg gA_1 \rightarrow A_2)\backslash\backslash A_3) \\
= & \{(17)\} \\
& (gA_1 \rightarrow A_1\backslash\backslash A_3) \,[\!]\, (\neg gA_1 \rightarrow A_2\backslash\backslash A_3) \\
= & \{(7)\} \\
& (A_1\backslash\backslash A_3) \mathbin{/\!/} (A_2\backslash\backslash A_3)
\end{aligned}
$$

Hence, $\backslash\backslash$ distributes over $/\!/$ to the right:

$$(\mathbf{A_1} \mathbin{/\!/} \mathbf{A_2})\backslash\backslash\mathbf{A_3} = (\mathbf{A_1}\backslash\backslash\mathbf{A_3}) \mathbin{/\!/} (\mathbf{A_2}\backslash\backslash\mathbf{A_3}) \tag{28}$$

4 Refinement

We now shortly present the refinement relation $\sqsubseteq$ between actions, discuss the relationship between ; and $\backslash\backslash$ with respect to refinement, as well as some monotonicity properties that are relevant for the dependency operator $\backslash\backslash$.

Refinement between actions We say that action A_1 is *refined* by action A_2, denoted $A_1 \sqsubseteq A_2$, if the weakest precondition of the former implies the weakest precondition of the latter, with respect to the same postcondition q, for all postconditions q:

$$A_1 \sqsubseteq A_2 \quad \text{iff} \quad \text{for all } q: \;\; wp(A_1, q) \Rightarrow wp(A_2, q) \tag{29}$$

The desired meaning of action refinement is that action A_2 is more deterministic than action A_1; this can be expressed as strengthening the guard ($gA_2 \Rightarrow gA_1$) and reducing non-determinism (e.g., $A_1 \,[\!]\, A_2 \sqsubseteq A_1 \mathbin{/\!/} A_2$).

Sequence is refined by dependency We have that $A_1 \,;\, A_2 \sqsubseteq A_1\backslash\backslash A_2$, based on the following calculations:

$$
\begin{aligned}
& wp(A_1 \,;\, A_2, q) \\
= & \{(19)\} \\
& wp([gA_1] \,;\, bA_1 \,;\, [gA_2] \,;\, bA_2, q) \\
= & \{(9),\ (10)\} \\
& gA_1 \Rightarrow wp(bA_1, gA_2 \Rightarrow wp(bA_2, q))
\end{aligned}
$$

$$
\begin{aligned}
& wp(A_1\backslash\backslash A_2, q) \\
= & \{(20)\} \\
& wp([gA_1] \,;\, [gA_2] \,;\, bA_1 \,;\, [gA_2] \,;\, bA_2, q) \\
= & \{(18),\ (10)\} \\
& gA_1 \wedge gA_2 \Rightarrow wp(bA_1, gA_2 \Rightarrow wp(bA_2, q))
\end{aligned}
$$

If we denote gA_1 by a, gA_2 by b, and $wp(bA_1, gA_2 \Rightarrow wp(bA_2, q))$ by c, we have to show that $(a \Rightarrow c) \Rightarrow (a \wedge b \Rightarrow c)$, which holds. Hence, we have that:

$$\mathbf{A_1} \,;\, \mathbf{A_2} \sqsubseteq \mathbf{A_1} \backslash\!\backslash \mathbf{A_2} \tag{30}$$

The reverse relation holds only if $gA_1 \Rightarrow gA_2$, hence:

$$\mathbf{A_1} \backslash\!\backslash \mathbf{A_2} \sqsubseteq \mathbf{A_1} \,;\, \mathbf{A_2} \quad \text{iff} \quad \mathbf{gA_1} \Rightarrow \mathbf{gA_2} \tag{31}$$

Monotonicity Nondeterministic choice $[\!]$ and sequential composition ; are monotonic with respect to refinement in both operands:

$$\mathbf{A_1} \sqsubseteq \mathbf{A_2} \wedge \mathbf{A_3} \sqsubseteq \mathbf{A_4} \quad \Rightarrow \quad \mathbf{A_1} \,[\!]\, \mathbf{A_3} \sqsubseteq \mathbf{A_2} \,[\!]\, \mathbf{A_4} \;\text{ and }\; \mathbf{A_1} \,;\, \mathbf{A_3} \sqsubseteq \mathbf{A_2} \,;\, \mathbf{A_4} \tag{32}$$

We first check the monotonicity of the dependency operator $\backslash\!\backslash$ in its left operand. Assume $A_1 \sqsubseteq A_2$:

$$\begin{array}{l}
\quad A_1 \backslash\!\backslash B \\
= \{(20)\} \\
\quad [gA_1] \,;\, [gB] \,;\, bA_1 \,;\, [gB] \,;\, bB \\
= \{\text{assumption properties}\} \\
\quad [gB] \,;\, [gA_1] \,;\, bA_1 \,;\, [gB] \,;\, bB \\
= \{(10)\} \\
\quad [gB] \,;\, A_1 \,;\, B \\
\sqsubseteq \{;\text{ is monotonic and } A_1 \sqsubseteq A_2\} \\
\quad [gB] \,;\, A_2 \,;\, B \\
= \{(20), (10)\} \\
\quad A_2 \backslash\!\backslash B
\end{array}$$

Hence, we have that the dependency operator is always monotonic in its left operand with respect to refinement:

$$\mathbf{A_1} \sqsubseteq \mathbf{A_2} \Rightarrow \mathbf{A_1} \backslash\!\backslash \mathbf{B} \sqsubseteq \mathbf{A_2} \backslash\!\backslash \mathbf{B} \tag{33}$$

When checking monotonicity for the dependency operator in the right operand, when $A_1 \sqsubseteq A_2$, we obtain that:

$$\begin{array}{l}
\quad B \backslash\!\backslash A_1 \\
= \{(20)\} \\
\quad [gB] \,;\, [gA_1] \,;\, bB \,;\, [gA_1] \,;\, bA_1 \\
= \{\text{assumption properties, } (10)\} \\
\quad [gA_1] \,;\, B \,;\, A_1 \\
\sqsubseteq \{;\text{ is monotonic and } A_1 \sqsubseteq A_2\} \\
\quad [gA_1] \,;\, B \,;\, A_2
\end{array}$$

$$\begin{array}{l}
\quad B \backslash\!\backslash A_2 \\
= \{(20),\ \text{assumption properties, } (10)\} \\
\quad [gA_2] \,;\, B \,;\, A_2
\end{array}$$

Hence, we have that the dependency operator is monotonic in its right operand with respect to refinement conditionally, namely if $gA_2 \Rightarrow gA_1$:

$$\mathbf{A_1} \sqsubseteq \mathbf{A_2} \Rightarrow \mathbf{B} \backslash\!\backslash \mathbf{A_1} \sqsubseteq \mathbf{B} \backslash\!\backslash \mathbf{A_2} \quad \text{iff} \quad \mathbf{gA_2} \Rightarrow \mathbf{gA_1} \tag{34}$$

5 A Train Example

We now present highlights from a larger case study on token movements on a trajectory. The trajectory is illustrated in Figure 3. Tokens can move in any direction and can entry and exit at any of the marginal segments, i.e, $\{L,\ B,\ G,\ M,\ N\}$. For simplicity we assume we only have one token in this paper, that needs to move from L to N. We discuss the $\mathscr{TS}$ Action System below, noting that it is inspired by a case study on train movements introduced in [1]. Here, we exclude traffic lights, switches and the possibility of one train occupying several segments, but add the possibility of loops. The $\mathscr{TS}$ action system can be thought of as a one-lane road system or communication network.

A token is an element under transportation in the network of slots connected to each other, realistically a packet, a car or anything with a 'reverse gear'. A token enters the network from a slot and exits the network from another slot, its destination. The token occupies at most one slot in the network at any given moment. The token is aware of its destination; this is a slot 'consuming' the token, like the IP-address in a TCP/IP packet or the end stop of a bus. Each slot is occupied by a token or is 'null', i.e., not occupied. Moreover, a non-empty subset of the slots may construct a loop that has a direction of looping, much like a roundabout.

In order for a token to advance from its point of entry towards its point of exit, dependency is crucial. Advancing from a slot to the next is an atomic action. Here the action relies on a token in the current slot but depends on the next slot not to be occupied, i.e. being in slot A and advancing to slot B is a situation where B's slot needs to be non-occupied, written $A\backslash\backslash B$. Thus, only if B can provide the service of admitting occupancy to the token, may the token be moved. When a token is advancing in the other direction, the dependency relation is, naturally, inversed: $B\backslash\backslash A$. For this, each slot is associated with two actions: fA, when a token is moved from slot A and tA, with a token is moved to slot A. Their corresponding guards are denoted gfA, gtA and their corresponding bodies are denoted bfA, btA, with f for "from" and t for "to".

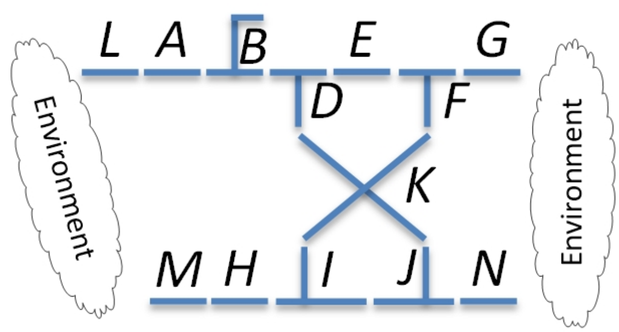

Figure 3: The example trajectory

Consider the set, $SLOTS$ and the string $token$; $SLOTS$ contains the names of the slots. We use the convention that slots are identified with capital letters and tokens with non-capital letters. We initialise these two sets as follows: $SLOTS = \{L, A, B, D, E, G, F, K, I,\ J, M, H, N\}$ and $token = a$. Here the token occupies the following slots in this order: $L, A, B, D, K, J, X*,\ N$, where X is the loop I, K, F, E, D, K, J and * stands for 0 to n times parsing the loop. Thus, the token may loop any number of times, but shall eventually exit the loop at J.

$$
\begin{array}{rcl}
\mathscr{TS} & = & |[\mathsf{var}\ SLOTS : \mathscr{P}(String),\ token : String, at : String \to String, \\
& & \quad dest : String \to String, neigh : \mathscr{P}(String \times String) \bullet \\
& & \quad SLOTS :=\{L,\ A,\ B,\ D,\ E,\ G,\ F,\ K,\ I,\ J,\ M,\ H,\ N\}\ ; \\
& & \quad token := a\ ;\ at :=\{L \mapsto a, A \mapsto null, ...\}\ ; \\
& & \quad dest :=\{a \mapsto N\}\ ;\ neigh :=\{(L,A),(A,L),...\} \\
& & \mathsf{do} \\
& & \ \ neigh(L,A) \to fL\backslash\backslash tA \\
& & []\, neigh(A,L) \to fA\backslash\backslash tL \\
& & []\, neigh(A,B) \to fA\backslash\backslash tB \\
& & []\, neigh(B,D) \to fB\backslash\backslash tD \\
& & []\, ... \\
& & \mathsf{od}\ ]|
\end{array}
$$

We have that $fX\backslash\backslash tY$ has the following form:

$$fX\backslash\backslash tY = (at(X) = token \to at(X) := null)\backslash\backslash(at(Y) = null \to at(Y) := token) \tag{35}$$

We notice that all choices are nondeterministic, implying that a token may loop forever instead of exiting. For this, the // operator may be used. Thus, we revise the loop exiting action as follows:

$$dest(token) = N \to (fJ\backslash\backslash tN)\ //\ (fJ\backslash\backslash tI)$$

Realistically, this means that if the token destination implies an exit, it will exit the loop if the next slot is not occupied.

6 Conclusions

In this paper we have initiated a study on a dedicated dependency operator, modeled via the Action Systems formalism. We have studied its commutativity, associativity, and distributivity over other composition operators, as well as its refinement rules. We have illustrated various concepts with several examples.

Action Systems is a state-based formalism, similar in that to Event-B [1], Z [29], Unity [13] (and its MobileUnity [24] extension), TLA [18], etc. One of the most popular such state-based formalisms is Event-B at the moment, justifiably based on the tool support as well as the refinement paradigm it promotes. Action Systems had an attempt at building a theorem prover tool, namely the Refinement Calculator [12], but that was a bit ahead of its time and was abandoned due to complexity and low interactivity. Action Systems are however highly flexible and versatile, also promoting modularity in a natural manner still not common to Event-B (but out of scope in this paper as well). Being able to model sequentiality, prioritised composition, dependency and reasoning about their properties is a clear advantage that Action Systems provide. Now, the next step is to be able to save all these operators and properties as a theory of actions, for instance via the theory plugin [19] in the Rodin platform, and being able to reuse such a theory in various contexts.

References

[1] J-R. Abrial, *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010. ISBN-13: 978-0521895569

[2] J. R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996. ISBN:0-521-49619-5

[3] J-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta and L. Voisin. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. In *International Journal on Software Tools for Technology Transfer (STTT)*, Vol. 12, No. 6, pp 447-466, Springer, 2010. doi:10.1007/s10009-010-0145-y

[4] R. J. Back. A Method for Refining Atomicity in Parallel Algorithms. In *E. Odijk, M. Rem, J.-C. Syre (eds), Proceedings of PARLE'89 – Parallel Architectures and Languages, Vol. 2: Parallel Languages*, pp. 199-216, 1989. doi:10.1007/3-540-51285-3_42

[5] R. J. Back. Refinement Calculus, Part II: Parallel and Reactive Systems. In *J. W. de Bakker, W.-P. de Roever, G. Rozenberg (eds), Proceedings of Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, Lecture Notes in Computer Science, Vol. 430. Springer-Verlag, 1990. doi:10.1007/3-540-52559-9_61

[6] R. J. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 131-142, 1983. doi:10.1145/800221.806716

[7] R. J. Back and R. Kurki-Suonio. Distributed Cooperation with Action Systems. In *ACM Transactions on Programming Languages and Systems*, Vol. 10, No. 4, pp. 513-554, 1988. doi:10.1145/48022.48023

[8] R. J. Back and K. Sere. Action Systems with Synchronous Communication. In *E.R. Olderog (ed), Proceedings of PROCOMET'94 – Programming Concepts, Methods, and Calculi*, pp. 107-126. IFIP Transactions A-56, North Holland, 1994.

[9] R. J. Back and K. Sere. From Action Systems to Modular Systems. In *Software - Concepts and Tools*, Vol. 17, pp. 26-39, Springer-Verlag, 1996. doi:10.1007/3-540-58555-9_83

[10] R. J. Back and K. Sere. Stepwise Refinement of Action Systems. In *J. L. A. van de Snepscheut (ed), Proceedings of MPC'89 – Mathematics of Program Construction*, pp. 115-138, 1989. doi:10.1007/3-540-51305-1_7

[11] M. Butler, E. Sekerinski, and K. Sere. An Action System Approach to the Steam Boiler Problem. In *J.-R. Abrial, E. Börger and H. Langmaack (eds), Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*. Lecture Notes in Computer Science, Vol. 1165, Springer-Verlag, 1996. doi:10.1007/BFb0027234

[12] M. Butler, J. Grundy, T. Långbacka, R. Ruksenas, and J. von Wright. The Refinement Calculator: Proof Support for Program Refinement. In *L. Groves and S. Reeves (eds), Proceedings of FMP'97 - Formal Methods Pacific*. Discrete Mathematics and Theoretical Computer Science Series, pp. 40-61, Springer-Verlag, 1997.

[13] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988. ISBN-13: 978-0201058666

[14] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall International, 1976. ISBN-13: 978-0132158718

[15] E. Hedman, J. N. Kok, and K. Sere. Coordinating Action Systems. In *D. Garlan and D. Le Métayer (eds), Proceedings of Coordination'97 – Coordination Languages and Models*, Lecture Notes in Computer Science, Vol. 1282, pp. 302-319, Springer-Verlag, 1997. doi:10.1007/3-540-63383-9_88

[16] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. In *Communications of the ACM*, Vol. 12, No. 10, pp. 576-580, 583, 1969. doi:10.1145/363235.363259

[17] C.A.R. Hoare. Communicating Sequential Processes. In *Communications of the ACM*, Vol. 21, No. 8, pp. 666-677, 1978.

[18] L. Lamport. The Temporal Logic of Actions. In *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 3, pp. 872-923, 1994. doi:10.1145/177492.177726

[19] I. Maamria, M. Butler, A. Edmunds, and A. Rezazadeh. On an Extensible Rule-based Prover for Event-B. In ABZ2010, Springer, 2010. doi:10.1007/978-3-642-11811-1_40

[20] Z. Manna and A. Pnueli. How to cook a temporal proof system for your pet language. In *Proceedings of the Tenth ACM Conference on Principles of Programming Languages*, pp. 141-154, ACM New York, 1983. doi:10.1145/567067.567082

[21] R. Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science, Vol. 92, Springer-Verlag, 1980. ISBN: 978-3-540-10235-9

[22] M. Neovius and K. Sere. Formal Modular Modelling of Context-Awareness. In *F. S. de Boer, M. M. Bonsangue and E. Madelain (eds), Formal Methods for Components and Objects, 7th International Symposium, FMCO 2008*. Lectures Notes in Computer Science, Vol. 5751, pp. 102-118, Springer-Verlag, 2008. doi:10.1007/978-3-642-04167-9_6

[23] L. Petre. *M*odelling with Action Systems. TUCS Dissertations No 69, November 2005. ISBN: 951-29-4018-3

[24] G.-C. Roman and P. J. McCann. A Notation and Logic for Mobile Computing. In *Formal Methods in System Design*, Vol. 20, No. 1, pp. 47-68, 2002. doi:10.1023/A:1012908529306

[25] M. Rönkkö, E. Sekerinski, and K. Sere. Control Systems as Action Systems - A Case Study. In *R. Smedinga, M.P. Spathopoulus, P. Kozák (eds), Proceedings of WODES'96 – Workshop on Discret Event Systems*, IEEE Press, pp. 362-367, 1996.

[26] E. Sekerinski. Deriving Control Programs by Weakest Preconditions. *TUCS Technical Reports*, No. 4, 1996.

[27] E. Sekerinski and K. Sere, A Theory of Prioritizing Composition. In *The Computer Journal*, Vol. 39, No 8, pp. 701-712. The British Computer Society, Oxford University Press, 1996. doi:10.1093/comjnl/39.8.701

[28] K. Sere and M. Waldén. Data Refinement of Remote Procedures. In *M. Abadi and T. Ito (eds.), Proceedings of TACS'97 – International Symposium on Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science, Vol. 1281, pp. 267-294, Springer-Verlag, 1997. doi:10.1007/PL00003935

[29] M. Spivey. *The Z Notation: A Reference Manual (Second Edition)*. Prentice Hall International Series in Computer Science, 1992.

[30] M. Waldén and K. Sere. Reasoning about Action Systems using the B-Method. In *Formal Methods in System Design*, No 13, pp. 5-35. Kluwer Academic Publishers, 1998. doi:10.1023/A:1008688421367