

Inter-service Dependency in the Action System Formalism

Extended Abstract

Mats Neovius^{1,2}, Fredrik Degerlund^{1,2}, Kaisa Sere¹

¹ Åbo Akademi University, Joukahaisenkatu 3 – 5, 20520 Turku, Finland

² Turku Center for Computer Science, Joukahaisenkatu 3 – 5, 20520 Turku, Finland
{mats.neovius, fredrik.degerlund, kaisa.sere}@abo.fi

1 Introduction

The use of formal methods is widely recognised for facilitating systematic construction of reliable and rigorous software. Methodologies supporting formalisation of functionality relying on distributed sources suggest to mastering inter-module dependencies where assignment of a global variable in one system changes the global state. The challenge comes to be identifying the properties and restrictions when formally defining dependencies involving the modules. The gain is that hence the systems need not to be administrated by a central entity and can be truly distributed. The modules can be independently replaceable as long as the functionality they guarantee remains intact given certain conditions. Consequently, the modules have well defined interfaces and they can be called services.

The motivation of our approach lies in that information sources become increasingly distributed; the information is provided by scattered independent entities [1, 2]. These entities depend on service(s) provide by other entities. A service can be an elementary source of information, some entity deducing new information depending on other (lower level) services or a combination of these two. Whether being an intermediate service or elementary source of information, the internal functionality of the provided service need not to be considered by the auxiliary services, i.e. the service can be considered a black-box.

The contribution of this extended abstract is in providing a glimpse into the research conducted in examining the means to formally rely on remote services and semantically reason about these. For this, we have defined an operator that masters the dependency relation that treats the remote services as stand-alone replaceable entities. Once mastering the formalisation of the services' interfaces, we claim that the formalism is ripe for specifying truly distributed inter-service dependent systems. We have chosen to model the dependency in the action system formalism framework, and we use reactive action systems as they provide means for reasoning about the information in a modular, distributed, manner. This extended abstract builds on our earlier work [3, 4].

2 Definitions of Concepts

For the reader to thoroughly understand this paper, some concepts need to be defined. We will consider systems that are either sources and/or utilisers of information. This paper will use *source* when indicating the origin of some information, realistically this is the input to the system or a sensor introducing some context. The *utiliser* constitutes the user of any provided information. Thus, an utiliser can be a source as well whenever it utilises other sources but changes these according to some rules such as calculating the mean value or by source introduction of its own. Realistically, this happens when a service depends on subservices to provide.

Because the source must not dictate its utiliser(s) but the utiliser selects the source(s), unidirectional dependencies are evident. Bidirectional dependencies in distinct traces are expressible, e.g. mutual agreement considering entities A and B where \hookrightarrow denotes “depends on”; $A \hookrightarrow B$ and $B \hookrightarrow A$. We will also model *direct* and *indirect* dependencies. In direct dependencies the utiliser will halt until the source provides its service whilst in indirect the utiliser settles for being guaranteed that the source will eventually execute the task. In addition, we will use other concepts specific to the formalism that are introduced gradually.

3. Characteristics of an action

One way of formally modelling software is to focus on the *state space* of a program. Each state in the state space is identified by the disjoint conditions that hold in it. Changes in these conditions are of central interest and are traced. Because the current state is well defined as are the executable tasks, a weakest precondition predicate can be derived for each task. Deriving one predicate from another one coins the idea of a predicate transformer, originally introduced by Dijkstra.

3.1. Actions at a glimpse

Since providing a table listing all possible preconditions for all post-execution states of an action is unmanageable, due to its sheer size, the approach taken is to describe this as a function describing the weakest precondition of an action [5]. The action system framework is a state based formalism for defining distributed systems [6, 7]. The basic component in an action system is the *action*. It bases on Dijkstra’s language of guarded commands [5, 8] and is defined with the *weakest precondition* predicate transformer, in short wp. From $wp(A, q)$ we can derive the weakest precondition, i.e. the conditions for which executing action *A* the postcondition *q* is satisfied. These pre- and postconditions are mere predicates over state variables. The weakest precondition is defined for various actions as follows:

$wp(\textit{magic}, q)$	$= \textit{true}$	<i>Miraculous action</i>	(1)
$wp(\textit{abort}, q)$	$= \textit{false}$	<i>Aborting action</i>	(2)
$wp(\textit{skip}, q)$	$= q$	<i>Stuttering action</i>	(3)
$wp(x := E, q)$	$= q[E/x]$	<i>Multiple assignment</i>	(4)

$$\text{wp}(A; B, q) = \text{wp}(A, \text{wp}(B, q)) \quad \textit{Sequential composition} \quad (5)$$

$$\text{wp}(A \square B, q) = \text{wp}(A, q) \wedge \text{wp}(B, q) \quad \textit{Nondeterministic choice} \quad (6)$$

$$\text{wp}([a], q) = a \Rightarrow q \quad \textit{Assumption} \quad (7)$$

$$\text{wp}(\{a\}, q) = a \wedge q \quad \textit{Assertion} \quad (8)$$

The action *abort* is used to model disallowed behaviour, thus q is never satisfied, i.e. the outcome is *false*. *skip* is a stuttering action, not doing anything, thus, the weakest precondition for establishing post-condition q is q . $x := E$ is multiple assignment where all occurrences of x are substituted with an element in E , $A; B$ is the sequential composition of two actions and $A \square B$ the (demonic) nondeterministic choice between actions A and B . $[a]$ is the assumption that is assumed true and $\{a\}$ is called the assertion that is a predicate needed to evaluate true in order for the execution to proceed to guarantee q . For assumption, if ‘ a ’ is *false*, the action behaves magically whilst for assertion, if ‘ a ’ evaluates false, the action aborts. Hence the actions *abort* and *magic* can be seen as special cases of assertion and assumption, respectively.

An action A is enabled ($gd A$) whenever executing it does not establish an unwanted post-condition.

$$gd A = \neg \text{wp}(A, \textit{false}) \quad \textit{Enabledness} \quad (9)$$

Hence, actions *abort*, *skip* and $x := E$ are always enabled.

This language allows guarded commands $[gA]; sA$, for convenience written $gA \rightarrow sA$, where gA is the guard. For the rest of the paper, we assume that any action A can be written in the form:

$$A = gA \rightarrow sA \quad \textit{Guarded command} \quad (10)$$

such that:

$$gd A = gA \quad (11)$$

and

$$gd sA = \textit{true} \quad (12)$$

Thus, enabledness of an action can be determined by checking its guard portion. Furthermore, we note that since $gd A = \textit{true}$, we can derive the following property of sA by applying the definition of enabledness (formula 9):

$$\text{wp}(sA, \textit{false}) = \textit{false} \quad \textit{Property } sA \quad (13)$$

Thereby sA must not establish a false post-condition. The weakest precondition semantics for an action $A = gA \rightarrow sA$ is:

$$\text{wp}(gA \rightarrow sA, q) = gA \Rightarrow \text{wp}(sA, q) \quad \textit{wp for guarded command} \quad (14)$$

Having defined the guarded actions, we can define conditional choice and repetitive construct:

$$\text{wp}(\textit{if } A \textit{ fi}, q) = \text{wp}(A, q) \wedge gA \quad \textit{Conditional choice} \quad (15)$$

$$\text{wp}(\textit{do } A \textit{ od}, q) = (\forall n. \text{wp}(A^n, gA \vee q)) \wedge (\exists n. \neg gA^n) \quad \textit{Repetitive construct} \quad (16)$$

where $A^0 = \textit{skip}$ and $A^{n+1} = A^n; A$. The repetitive construct defines that each action enables some action or establishes q and that there must exist some that does not enable any other, i.e. partial correctness and termination. Consequently, an action A within **do** ... **od** may execute only when its guard gA holds.

3.2 Inter-action Dependencies

Expressing that an action depends on another action can be modelled using a special operator. We denote it the *dependency operator* $\backslash\backslash$. Letting A and B be actions, where $A \backslash\backslash B$ denotes that $A \hookrightarrow B$, we define $\backslash\backslash$ to be:

Def. 1, dependency operator: $A \backslash\backslash B = gA \wedge gB \rightarrow A; B$

Whenever having the construct $A \backslash\backslash B$, we call the dependent action A the *native* action and B the *trailing* action. For the gB to evaluate true after having executed A , we need to assure that A preserves gB by not assigning the free variables of B so that it would disable gB . This characteristic is shown in Section 3.3 by calculating the guard for $A \backslash\backslash B$, as is the wp for $A \backslash\backslash B$. Typically, at the time of termination $A \backslash\backslash B$ would be disabled, i.e. modelled to be executed exactly once.

3.3 Characteristics of the $\backslash\backslash$ -operator

The dependency operator introduces some restrictions. We examine these by defining how $\backslash\backslash$ relates to the provided semantics of section 3.1 by exposing its characteristics by calculating its weakest precondition.

Characteristic 1, wp for $\backslash\backslash$: $\text{wp}(A \backslash\backslash B, q)$
 $= \text{wp}(gA \wedge gB \rightarrow A; B, q)$ // Def. 1
 $= gA \wedge gB \Rightarrow \text{wp}(A; B, q)$ // formula 14
 $= gA \wedge gB \Rightarrow \text{wp}(A, \text{wp}(B, q))$ // formula 5
 $= \text{wp}(gA \wedge gB \rightarrow A, \text{wp}(B, q))$ // rewrite 14

This characteristic coins the meaning of the $\backslash\backslash$ operator. Initially gA and gB need to hold and after executing A , a state where B is enabled is reached, and after executing B , q is established. Hence, A must not disable B .

Assuming that B establishes q whenever gB holds and executed after A , we can calculate the collective guard of $A \backslash\backslash B$. This collective guard is deduced with the help of wp formulae.

Characteristic 2, guard of $\backslash\backslash$: $g(A \backslash\backslash B)$
 $= \neg \text{wp}(A \backslash\backslash B, \text{false})$ // formula 8
 $= \neg \text{wp}(gA \wedge gB \rightarrow A, \text{wp}(B, \text{false}))$ // charac. 1
 $= \neg(gA \wedge gB \Rightarrow \text{wp}(A, \text{wp}(B, \text{false})))$ // formula 14
 $= \neg(gA \wedge gB \Rightarrow \text{wp}(A, \text{wp}(gB \rightarrow sB, \text{false})))$ // formula 10
 $= \neg(gA \wedge gB \Rightarrow \text{wp}(A, gB \Rightarrow \text{wp}(sB, \text{false})))$ // formula 14
 $= \neg(gA \wedge gB \Rightarrow \text{wp}(A, gB \Rightarrow \text{false}))$ // formula 13
 $= \neg(gA \wedge gB \Rightarrow \text{wp}(A, \neg gB \vee \text{false}))$ // Def. \Rightarrow
 $= \neg(gA \wedge gB \Rightarrow \text{wp}(A, \neg gB))$ // tautology
 $= \neg(\neg(gA \wedge gB) \vee \text{wp}(A, \neg gB))$ // Def. \Rightarrow
 $= \neg\neg(gA \wedge gB) \wedge \neg \text{wp}(A, \neg gB)$ // Def. deM
 $= gA \wedge gB \wedge \neg \text{wp}(A, \neg gB)$ // double neg

Characteristic 2 defines the general structure of the guard that must hold for the action $A \backslash\backslash B$ to be enabled. In short, the gA and gB need to hold and A must not disable B .

$A \backslash\backslash B$ is not commutative. This is the case partly because the significance of order i.e. distinction between the native action and trailing action in definition 1, where the native

action must not disable the trailing one. These characteristics support the definition provided. Hence, we conclude property 1 for non-interference:

Property 1, non-interference native: The native action can only assign the free variables of the trailing action in a manner that does not disable the guard of the latter.

Property 1 states that a native action A must assure not to contribute towards disabling the trailing action B . Therefore, A is not allowed to arbitrarily assign the variables of B . However, the trailing action B can assign the native action's variables; otherwise, the impact of the trailing action would be restricted to the inclusion of the guard of the dependency relation.

4. Dependency on an action system level

As the fundamentals of an action and the characteristics of the dependency operator are provided, we extend usage of the operator to be used within an action system. The action system building blocks are defined in Section 4.1. We consider reactive action systems, where independent systems operate as a part of a more complex system. To navigate the complex system and depend on these remote action systems, we introduce a means for remote referencing in Section 4.2.

4.1. Action system at a glimpse

To start reasoning with action systems, we specify the elements of one, here named \mathcal{A} :

Def. 4, action system: $\mathcal{A} = \llbracket [\text{var } v, w^* \text{ proc } P:p; R^*:r \bullet \text{Init}: A_0; \text{do } Op: A \text{ od}] \rrbracket : i$

In \mathcal{A} , v and w^* are the variables declared by this action system. Variables v are *local* and w^* constitute the *exported* variables (denoted with an asterisk). Procedures are declared in the clause *proc* where P : p is a local procedure p named P , only executed if called upon whilst R^* is a globally referable procedure. Action $\text{Init}:A_0$ is the initialising action assigning the declared variables their initial value where Init is the label of this action. Each action label $\in \text{Name of action labels}$ in the declaring action system. The **do ... od** bracket pair constitute the repetitive construct (formula 16) within which the action A labelled Op is repeatedly executed until A aborts or until termination. Variables i stand for the optional *imported* variables that are declared and exported by other action systems but referenced from this. Together, import i and export w^* constitute a situation resembling shared writable memory where the variable type is declared by the exporting action system.

Because considering reactive action systems the action system \mathcal{A} is a part of a more complex system, where all other action systems are considered as \mathcal{A} 's environment, commonly denoted as \mathcal{E} . As the action atomicity holds on the whole complex system, any atomic action of \mathcal{A} can be preceded by an action in \mathcal{E} impacting \mathcal{A} by writing to \mathcal{A} 's global variable space. Hence, the reactive component does not terminate by itself as the environment can, through the global variables, enable some actions within this. This makes the termination a global property and the formalism comes to showing properties of execution traces.

Any set of action systems in the reactive system can be composed to form a coherent monolithic action system. This is realised with the commutative and associative parallel composition operator \parallel , defined in Definition 2:

Def. 2, parallel composition ‘ \parallel ’: Let

$\mathcal{A} = \llbracket \mathbf{var} \ v_a, w_a^*; \mathbf{proc} \ P:p \bullet \mathbf{Init}:A_0; \mathbf{do} \ Op:A \ \mathbf{od} \rrbracket : i$ and

$\mathcal{B} = \llbracket \mathbf{var} \ v_b, w_b^*; \mathbf{proc} \ R^*:r \bullet \mathbf{Init}:B_0; \mathbf{do} \ Op:B \ \mathbf{od} \rrbracket : j$ then

$\mathcal{C} = \mathcal{A} \parallel \mathcal{B} = \llbracket \mathbf{var} \ x_m, x_n^*; \mathbf{proc} \ P:p, R^*:r \bullet \mathbf{Init}:A_0; B_0;$

$\mathbf{do} \ Op\mathcal{A}: A \ [] \ Op\mathcal{B}: B \ \mathbf{od} \rrbracket : h$ where

$h = i \cup j \setminus (w_a \cup w_b), x_n^* = w_a \cup w_b$ and $x_m = v_a \cup v_b$ provided that $v_a \cap v_b = \emptyset$.

Definition 2 states that if a set of action systems operates on a disjoint set of local variables, $v_a \cap v_b = \emptyset$, procedure names and action labels, they can be composed without renaming to one action system where the actions within the repetitive **do** ... **od** loop are treated non-deterministically and procedures remain intact. If the local variables are not disjoint or the local procedure names coincide; non-overlapping can be achieved through renaming whereas the action labels are given a suffix indicating their origin. In the declaration above, action system \mathcal{C} is a parallel composition of \mathcal{A} and \mathcal{B} where the possible execution traces remain unchanged. \parallel has the immediate drawback of compromising modularity and reusability, i.e. composing action systems $\mathcal{A} \parallel \mathcal{B} = \mathcal{C}$ does not guarantee that once decomposing \mathcal{C} , \mathcal{A} and \mathcal{B} are recovered in their original form. Consequently, composition provides a means to form an abstract view of the system as well as refactoring the system.

4.2. Inter-Action System dependencies

Dependency within one action system is denoted by applying the dependency operator within the **do** ... **od** construct with a reference to an action within this same repetitive construct. For referring to actions in the environment of this action system, means to make “remote dependency references” need to be defined. The trailing action operates in its own right, i.e. possibly providing its service to many disjoint actions without these having to be aware of each other. To reference a remote system providing this service, we define the $@$ reference:

Def. 3, @ reference: Let B be an action and $\mathcal{B}ar$ an action system where $B \in \text{actions of } \mathcal{B}ar$, then $B@B\mathcal{a}r$ refers to action B in action system $\mathcal{B}ar$.

The $@$ is a postfix to an action where $A \setminus B@B\mathcal{a}r$ denotes action A to depend on action B in action system $\mathcal{B}ar$. Because of the atomicity of \setminus , action $A \setminus B@B\mathcal{a}r$ waits for B to finish; calling this a *direct dependency* relation. Consequently, several actions can depend on $B@B\mathcal{a}r$ without interference as $B@B\mathcal{a}r$ is atomic and provides only to one system at any given time resembling the situation where A requests a resource possessed by $B@B\mathcal{a}r$.

Direct dependency relations do however halt the execution of the native system until the referenced action $B@B\mathcal{a}r$ terminates. As the scheduling for the whole system is not of interest, breaking the atomicity might be of interest. This can be done whenever $B@B\mathcal{a}r$ only enables another action, labelled $B_{\text{wake}}@B\mathcal{a}r$ that ought to be executed in the wake of A . Hence, if $B@B\mathcal{a}r$ enables B_{wake} and the system can guarantee that B_{wake} is executed at some point after the reference, the atomicity of \setminus is broken. This lets the native system continue its execution until the possible results of B_{wake} are required. Thus,

$A \setminus B @ Bar$ only enables B_{wake} that is assured to execute prior to the first execution of the action labelled B_{nat} .

$$\begin{aligned} \mathcal{A} &= \llbracket \mathbf{var} \ v, w^* \ \mathbf{proc}; \bullet \ \mathbf{Init}: A_0; \ \mathbf{do} \ Op: gOp \rightarrow A \setminus B @ Bar \\ &\quad \llbracket \text{“other actions”} \ \mathbf{od} \rrbracket : i \\ Bar &= \llbracket \mathbf{var} \ j, i^* \ \mathbf{proc} \ B^*: gB_{orig} \wedge k = false \rightarrow k := true; \bullet \ \mathbf{Init}: Bar_0; \\ &\quad \mathbf{do} \ B_{wake}: k = true \wedge gB_{orig} \rightarrow sB_{orig}; \ C; \ k := false \\ &\quad \llbracket B_{nat}: k = false \wedge gB_{orig} \rightarrow sB_{orig} \\ &\quad \llbracket \text{“other actions”} \ \mathbf{od} \rrbracket : l \end{aligned}$$

Here actions labelled B_{nat} and B_{wake} assure sB_{orig} to be executed once but in addition to sB_{orig} , the referenced action labelled B_{wake} executes an additional action (possibly stuttering) C and disables itself through assigning k false. As the guard of the globally referable procedure B^* is the same as for the alternatives of B_{nat} and B_{wake} , the semantics of the remote dependency is not altered. The impact on the system is similar to the one when considering only intra-action system dependencies but the atomicity is deliberately broken down with the Boolean of k for the sake of immediate progress, i.e. being able to execute the “other actions” in \mathcal{A} before B_{wake} is finished. We call this *indirect dependency*. Moreover, if action system \mathcal{A} is the only system importing variable k , then we say that k is a dedicated variable for this dependency. The trade-off with breaking the dependency is that the execution order cannot be guaranteed and it should hence be used carefully, i.e. as above when B^* enables B_{wake} , $A \setminus (B \setminus C)$ ordering is not necessarily kept as $A; B_{wake}; C$ because B_{wake} might actually execute after C , which is obvious as the atomicity was deliberately broken. However, it can easily be shown that A executes before C and before B_{wake} .

5. A short example

To clarify the realistic implementation scope of the ideas presented in this extended abstract, we outline a short, easily conceivable example. This example bases on fraction of a Buyer-Seller relation where the seller runs an ERP-system (Enterprise Resource Planning).

Assuming two actions $A \hookrightarrow B$, a realistic scenario could be that A wants to buy something sold by B i.e. a normal buyer-seller relation. Letting gA be ‘has money’ and A constitute the state update, gB could realistically be ‘in stock’ where B merely updates the stock. Consequently, per Definition 1, $gA \wedge gB \rightarrow A; B$, buyer A has money whilst the product is in stock and once bought the stock is updated. The novelty of this approach is that in order for A to execute, gB needs to be true, i.e. product must be in stock for the buyer to buy¹ and A need only to know the “interface” of B , i.e. sell if in stock.

$$\begin{aligned} Buyer &= \llbracket \mathbf{var} \ v, w^* \ \mathbf{proc}; \bullet \ \mathbf{Init}: A_0; \ \mathbf{do} \ Buy: gA \rightarrow A \setminus B @ Seller \ \mathbf{od} \rrbracket : i \\ Seller &= \llbracket \mathbf{var} \ j, i^* \ \mathbf{proc}; \bullet \ \mathbf{Init}: B_0; \ \mathbf{do} \ B: gB \rightarrow B \ \mathbf{od} \rrbracket : l \end{aligned}$$

However, as there is no reason why the *Buyer* should halt until the *Seller* is complete, we break the atomicity.

$$Buyer = \llbracket \mathbf{var} \ v, w^* \ \mathbf{proc}; \bullet \ \mathbf{Init}: A_0; \ \mathbf{do} \ Buy: gA \rightarrow A \setminus B @ Seller \ \mathbf{od} \rrbracket : i$$

¹ Mathematically, knowing this a priori is irrelevant as the system can be modelled so that once the purchase is done, the product must be in stock.

$$Seller = \llbracket \text{var } j, i^* \text{ proc } B^*: gB \wedge j = false \rightarrow j := true; \bullet \text{ Init: } B_0; \\ \text{do } B_{wake}: gB \rightarrow B; D \\ \llbracket B_{nat}: gB \rightarrow B \text{ od} \rrbracket : l$$

Here the *Seller's* gB stand for “in stock”. Hence, the dependency relation in the action labelled *Buy* is disabled unless gB evaluates true. The auxiliary action D executes only in the wake of a dependency reference to $\text{proc } B^*$ and could stand for shipping the product and invoicing. Naturally, this relation can be extended to a multi-phased dependency relationship where payment, cancellation, trust and other policies can be included.

6. Conclusions

As the future is likely going to be about navigating the ubiquity of information, being able to select, rely on and process relevant information, as well as to reason rigorously with these, the need to formally treat remote providers of this information is evident. In this extended abstract, we have shown research results in how inter-service dependencies can be modelled in the action system framework. We have introduced and briefly outlined the characteristics of a new dependency operator \llbracket and exemplified its feasibility in a short example.

As the research on the \llbracket operator is far from finished, future work will address chains of dependencies, non-ordered dependencies as well as refinement. The goal is to gain and present a library of well defined operators that address the challenges brought along with the ever increasing distribution of computations and responsibilities. We believe the service oriented architecture provides a feasible and demanding platform to verify results upon.

References

1. Neovius, M. and Yan, L.: A Design Framework for Wireless Sensor Networks. In Proceedings of World Computer Congress - WCC 2006, Ad Hoc networking track, 2006.
2. Roman, G.C., Julien, C., and Payton, J.: A formal treatment of context-awareness, In Proceedings of FASE'04, 2004.
3. Neovius, M. and Sere, K.: Formal Modular Modelling of Context-Awareness. To appear in Post-proceedings of FMCO 2008. LNCS, 2009.
4. Degerlund, F. and Sere, K.: A Framework for Incorporating Trust into Formal Systems Development. In Theoretical Aspects of Computing – ICTAC 2007, 4th International Colloquium, Proceedings, 2007.
5. Dijkstra, E. W.: A Discipline of Programming. Prentice Hall, 197632.
6. Back, R.J.R and Kurki-Suonio, R.: Decentralization of Process Nets with Centralized Control. In Proceedings of the 2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing, 1983
7. Sere, K.: Stepwise derivation of parallel algorithms, PhD dissertation, Åbo Akademi 1990.
8. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM, vol. 18, no. 8, 1975.