Inter-service Dependency in the Action System Formalism

Kaisa Sere

Åbo Akademi University, Finland

Joint work with

Mats Neovius

and

Fredrik Degerlund

Overview of the talk

- **Part I: Basics of Action Systems**
  - *Schematic view*
  - *Weakest preconditions*
  - *Stepwise refinement*
  - *etc.*
- **Part II: Modelling Services in the Action System Formalism**
  - *General approach*
  - *Dependency operator*
  - *Contract-based interface*
  - *etc.*

Åbo Akademi University

Part I

Basics of Action Systems

Åbo Akademi University

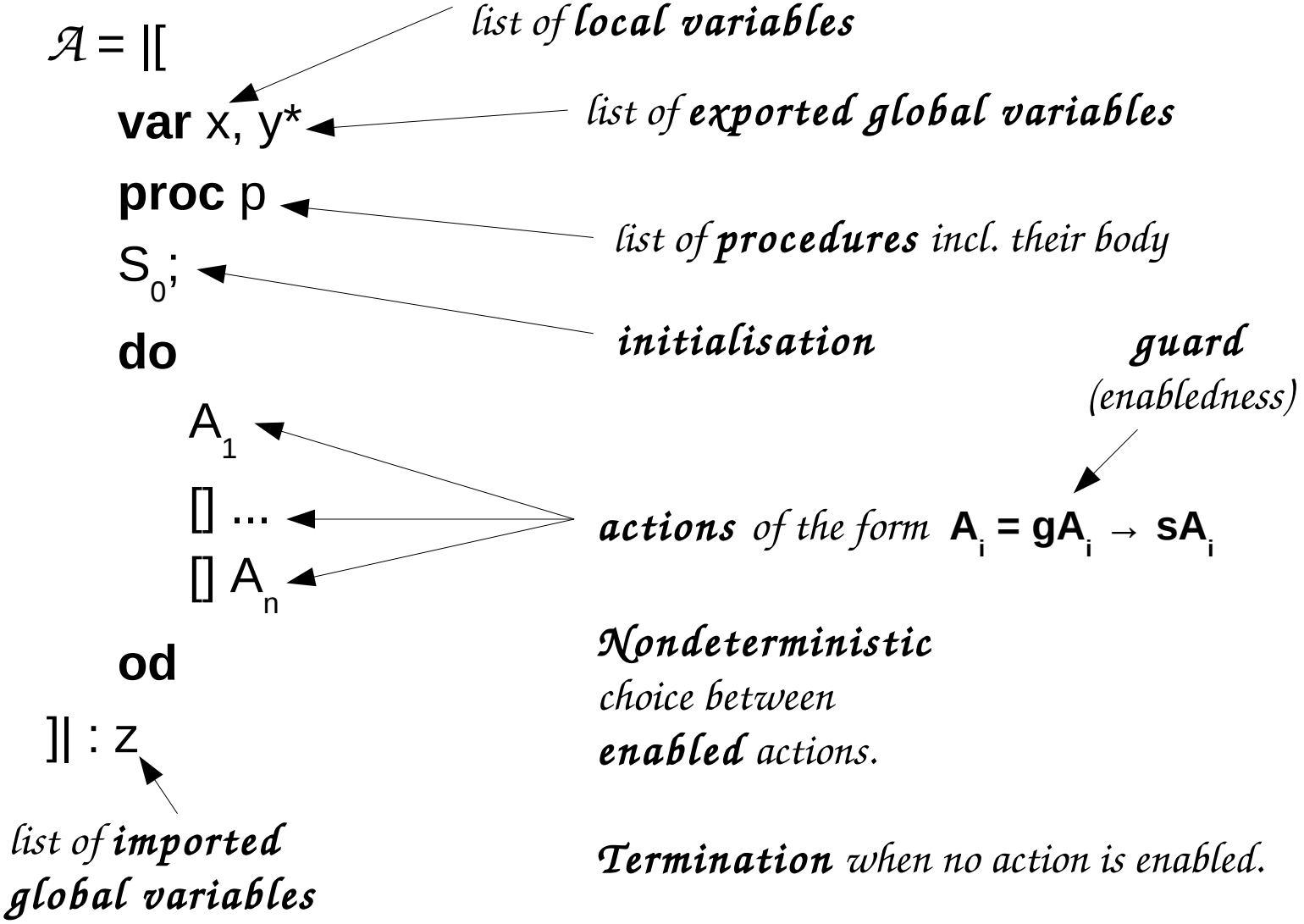
Why use formal methods?

- Formal methods provide a means of proving *correctness* of programs.
  - Testing alone cannot guarantee the non-existence of flaws in non-trivial programs.
- Different types of formal methods.
  - *Program refinement*
    - Stepwise derivation from initial specification.
  - *Model checking*
    - Proving properties about a model.
- Our approach is based on refinement.

The action system formalism

- Originally proposed by R.J.R. Back and R. Kurki-Suonio.
  - Has been extended by several contributors.
- Supports stepwise refinement.
- Based on E.W. Dijkstra's guarded command language.
  - Weakest precondition semantics.
    - Predicate transformers.

Schematic view of an action system



Weakest precondition predicate transformers

- *Predicate.*
  - A boolean function from the state space.
- *Predicate transformer.*
  - A higher order functions, mapping predicates to predicates.
- *Weakest precondition predicate transformer.*
  - wp(*A*, *q*) is the weakest precondition predicate transformer for action *A*, returning a predicate evaluating *true* exactly in the states in which executing A will establish predicate *q*.

List of fundamental wp's

- $\mathrm{wp}(\mathit{magic}, q) = \mathit{true}$
- $\mathrm{wp}(\mathit{abort}, q) = \mathit{false}$
- $\mathrm{wp}(\mathit{skip}, q) = q$
- $\mathrm{wp}(x{:=}E, q) = q[E/x]$
- $\mathrm{wp}(A \,[]\, B, q) = \mathrm{wp}(A, q) \wedge \mathrm{wp}(B, q)$
- $\mathrm{wp}(A; B, q) = \mathrm{wp}(A, \mathrm{wp}(B, q))$
- $\mathrm{wp}([a], q) = a \Rightarrow q$
- $\mathrm{wp}(\{a\}, q) = a \wedge q$

Guards

- In the **do**-**od** loop, each action is of the form:

  A = gA → sA
- This is a short-hand notation for:

  A = [gA]; sA
  - gA is called the guard (of the action).
  - sA is called the statement (of the action).
  - [...] is an assumption.
    - wp([*a*], *q*) = *a* ⇒ *q*
  - ";" indicates sequential composition.
    - wp(*A*; *B*, *q*) = wp(*A*, wp(*B*, *q*))

Extracting the guard

- The guard can be computed as:
  - g(*A*) = ¬wp(*A*, *false*)

- We assume that for each guarded command A = gA → sA, the following holds:
  - g(*A*) = *gA*
  - g(*sA*) = *true*
  - This means that the guard of a guarded command can easily be identified as its gA part.

Enabledness

- Each action is of the form:

  A = gA → sA
- This is a short-hand notation for:

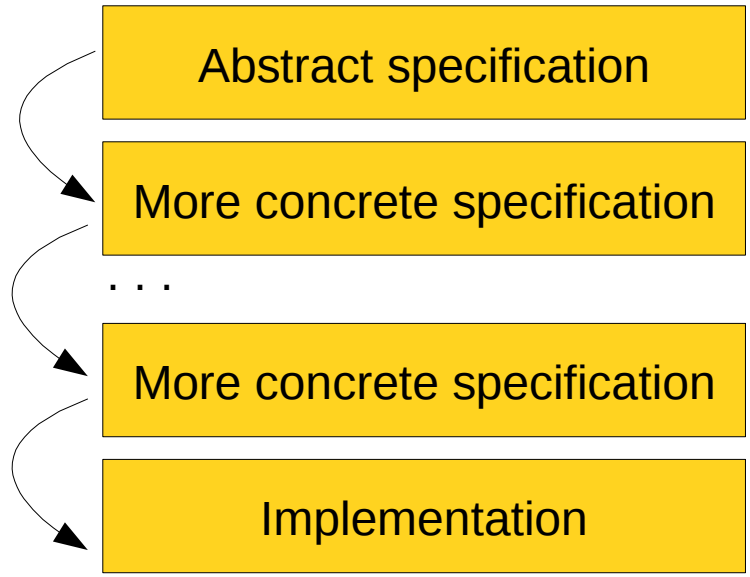
  A = [gA]; sA
  - gA is called the guard (of the action).
  - sA is called the statement (of the action).
  - [...] is an assumption.
    - wp([*a*], *q*) = *a* ⇒ *q*
  - ";" indicates sequential composition.
    - wp(*A*; *B*, *q*) = wp(*A*, wp(*B*, *q*))

Refinement

- Definition: Refinement of actions.
  - $A \sqsubseteq A' \Leftrightarrow \forall q.\ \mathrm{wp}(A, q) \Rightarrow \mathrm{wp}(A', q)$
- Refinement of action systems.
  - Refinement w.r.t. input-output.
    - Total correctness
  - Trace refinement.
    - Interest in intermediate, observable states.
    - Needed in our work.

Stepwise refinement chain

1) Start from an initial, abstract specification.
2) Rewrite into a more concrete specification.
3) Prove the correctness of the new specification w.r.t. the previous one.
4) If not yet concrete enough (implementation) go to step 2.



Åbo Akademi University

Part II

Modelling Services in the Action System Formalism

Our current goals / challenges

- Avoid modelling systems as a monolith.
- Encouraging reusable and replacable modules.
- Components can be treated as services.
- A contract-based interface between the utilising entity and the service(s).

Concepts

- *Source.*
  - An entity constituting the origin of some information.
- *Utiliser.*
  - An entity using information provided by a source.
  - Can also be a source itself.
- *Dependency.*
  - A utiliser is said to be dependent on a source if it needs said source in order to provide its functionality.

Expressing dependencies

- We introduce a special dependency operator for expressing dependencies.
- Definition: *Dependency operator.*
  - A\\B = gA ∧ gB → A; B
- Can also be expressed as:
  - A\\B = gA ∧ gB → (sA; gB → sB)
- Intuition: In A\\B, A is an entity depending on another entity, the service B, in order to provide its functionality.
  - A constitutes a utiliser.
  - B constitutes a source.

Weakest precondition for \\

- wp for \\ can be derived using fundamental wp formulae.

$$\mathbf{wp(A \backslash\backslash B, q)}$$
$$= \mathrm{wp}(gA \wedge gB \rightarrow A; B, q)$$
$$= gA \wedge gB \Rightarrow \mathrm{wp}(A; B, q)$$
$$= gA \wedge gB \Rightarrow \mathrm{wp}(A, \mathrm{wp}\ (B, q))$$
$$\mathbf{= wp(gA \wedge gB \rightarrow A, wp(B, q))}$$

Guard of \\

g(*AB*)**

$= \text{wp}\ (A \backslash\backslash B, \mathit{false})$

$= \neg\text{wp}\ (gA \wedge gB \rightarrow A, \text{wp}\ (B, \mathit{false})$

$= \neg(gA \wedge gB \Rightarrow \text{wp}\ (A, \text{wp}\ (B, \mathit{false}))$

$= \neg(gA \wedge gB \Rightarrow \text{wp}\ (A, \text{wp}\ (gB \rightarrow sB, \mathit{false})))$

$= \neg(gA \wedge gB \Rightarrow \text{wp}\ (A, gB \Rightarrow \text{wp}\ (sB, \mathit{false}))$

$= \neg(gA \wedge gB \Rightarrow \text{wp}\ (A, gB \Rightarrow \mathit{false}))$

$= \neg(gA \wedge gB \Rightarrow \text{wp}\ (A, \neg gB \vee \mathit{false}))$ *// def. ⇒*

$= \neg(gA \wedge gB \Rightarrow \text{wp}\ (A, \neg gB))$ *// tautology*

$= \neg(\neg(gA \wedge gB) \vee \text{wp}\ (A, \neg gB))$ *// def. ⇒*

$= \neg\neg(gA \wedge gB) \wedge \neg\text{wp}\ (A, \neg gB)$ *// deMorgan*

$= \mathbf{gA \wedge gB \wedge \neg wp\ (A, \neg gB)}$ *// double neg*

Example

$\mathcal{A}$ = |[

var x

proc utiliser = {...},

service = {...}

S_0;

do

utiliser \\ service

[] ...

od

]|

Direct vs indirect dependencies

- *Direct / hard dependencies.*
  - Utiliser and service executed as an atomic entity.
  - Easily expressed using the \\ operator.
- *Indirect / soft dependencies.*
  - Utiliser executed first, then possibly other actions. Service is guaranteed to be executed at some point.

Contracts

- Defines the interface between utiliser and source.
- Source defines a contract.
  - Utiliser must accept it in order to use the service of the source.
- General constraints:
  - The utiliser must not write on the source's variables in such a way that the latter becomes disabled.

Contracts continued

- The *general* constraints can be expressed as follows:

gA ∧	**gB →**	**sA ;**	**gB →**	**sB**
R util var	R util var	R util var	R util var	R util var
R serv var	R serv var	R serv var	R serv var	R serv var
		W util var		W util var
		W $serv_x$ var		W serv var

- R = *read*, W = *write*
- util var / serv var = *utiliser's / service's var*
- W $serv_x$ var = *write only in such a way as not the disable the service*

Conclusions

- Presented a framework for expressing dependencies / use of services in action systems.
- Introduced a new dependency operator.
- Explored properties of the dependency operator.
- Interface between utiliser and source is contract based.
- Indirect dependencies are an alternative to direct, atomic dependencies.

Recent & future work

- We have recently submitted a conference paper on the topic.
- More closely explore separation not only into utiliser/source, but also into separate action systems.
- More research into refinement rules for dependencies.
- Explore indirect dependencies (soft dependencies) more closely.

Åbo Akademi University

Thank you!