Formal Analysis of a Local Segmented Bus Arbiter

Tomi Westerlund
Turku Centre for Computer Science
Turku, Finland
tomi.westerlund@utu.fi

Tiberiu Seceleanu
University of Turku
Turku, Finland
tiberiu.seceleanu@utu.fi

Abstract

In this paper we show the derivation of a local segmented bus arbiter from a single segment bus arbiter. The arbiter is extended to cater for requests coming from and going to external segment. The derivation demonstrates the capability of preserving correctness when considering an important hardware design decision. The operations are performed in the formal framework of Action Systems. Action Systems is a predicate transformer based method for modelling reactive systems. It has also been successfully applied to both asynchronous and synchronous hardware designs.

1 Introduction

The latest advances in technology allowed more and more functionality to be placed within a single chip. The complexity of such systems comes as an inherent byproduct, which leads further to problems concerning the correctness of the development flow. On one hand, modular design is one of the solutions towards partially reducing the task of the designer of complex systems, while on the other hand, the employment of formal methods in system design tries to solve the aspects related to correctness. However, the pressure imposed by time-to-market usually does not leave enough time for thorough analysis of designs before they are shipped. Additionally, the often heavy mathematical apparatus behind formal method frameworks still forbids the expected wide usage of such approaches in (hardware) system design.

The traditional approach to bus designs relies on a single segment bus. Due to multiple reasons, this may not continue to serve well the design flow of highly integrated systems. We face the problem in adapting existent designs to the new environment. In the following sections, we address this issue by showing how a previous bus arbiter description can be correctly transformed into a local segment bus arbiter. For this, we apply action systems technique. Action systems is a state-based formalism, relying on an extended version of Dijkstra's language of *guarded commands* [1].

2 The Segmented Bus Platform

Modern deep sub-micron silicon technologies have given a real boost to system-on-chip (SOC) design research and development. The growing diversity of devices brings up an immense number of possible interfaces. In many situations, both the system design and performance are limited by the complexity of the interconnection between the different modules and blocks that are integrated into those chips. Furthermore, different data transfer speeds are required as well as parallel transmission. A simple bus is one such structure that may not be suitable for such a design. This is because only one module can transmit at a time and the bus is slow due to large capacitive load caused by the interfaces of the modules that are attached to it and the long physical length.

A solution to the above mentioned problems is a segmented bus design combined with a globally asynchronous locally synchronous (GALS) [2] system architecture. In this approach, each module of a SOC system is synchronised to a local optimised clock whereas interactions between those modules are arranged asynchronously. A segmented bus [3] is a bus which is partitioned into a segments. Each segment act as a normal bus by themselves. These segments can be connected dynamically to each other to form a larger bus structure. Due to the segmentation of this resource, parallel transaction can take place, thus increasing the performance.

2.1 Segmented Bus Architecture

The segmented bus structure is simply illustrated in figure 1. Every *segment* is composed of masters, slaves, a local arbiter, the physical lines (address, data, request, acknowledge and read / write lines), and an *inter-segment bridge* controller as shown in figure 2. Most of the time masters are asking services from local slaves. Occasionally, one master may require services from a slave residing in some other segment. In this situation, the local arbitration unit forwards the request to a central arbitration module.

The central arbiter (CA) stores the information regarding the current situation of the segments: what segments are participating in an inter-segment transaction, and what segments are requesting for an inter-segment access. Based on this information, the CA informs a change in the ownership of the segments that are participating an inter-segment transfer.

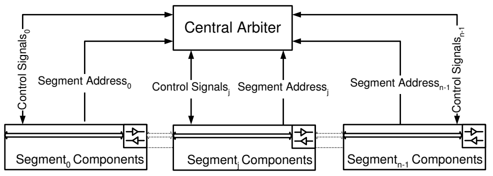


Figure 1: Segmented bus structure.

In this paper we concentrate on the description of the segment arbiter (SA), the one that coordinates the activity within one segment limits.

2.2 Operations on a Segmented Bus

There are three modes in which operation on a specific segment may proceed, from the point of view of local arbitration. These modes depend on the localisation of the master requesting the bus and the slave. Thus, we have (i) a *local* *master – local slave*, (ii) a *local master – external slave* and (iii) a *external master – local or external slave* situation.

In all the situations, the master that is granted the access to the bus connects to the slave following a four-phase signalling protocol. The *request* part is also visible to the SA residing in the same segment as the master. Thus, the SA supervises the access of the master to the bus by counting the number of transfers, in cases (i) and (ii) above.

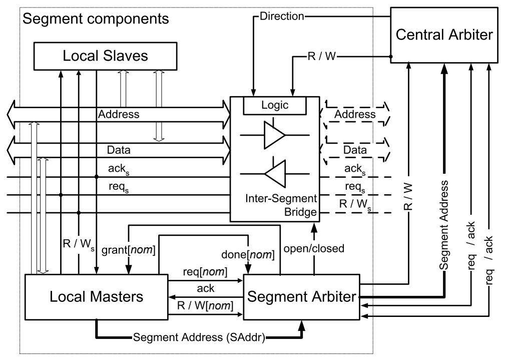


Figure 2: The Segment Control Elements.

All the participants in the segmented bus implementation are mutually asynchronous devices. Therefore, the communication follows handshake protocols. The only situation where the handshake protocols are not reflected are related to the logic, which controls the bridges. In this case, the segment arbiter only sends the command to open or close the bridge, while the direction is specified by the CA.

3 Action Systems

Back and Kurki-Suonio [4] introduced the action systems formalism, providing a framework for specifying and refining concurrent programs. An *action system* (AS) is in general a collection of *actions* or guarded commands, which are executed one at a time. The Action Systems is used for specification and correctness preserving development of reactive systems. It was first tailored to a software system design but is then successfully applied also to a hardware system design, both synchronous [5] and asynchronous [6].

An *action* A is defined (for example) by

$$\begin{array}{lll} A ::= & x := x'.R & \text{(nondeterministic assignment)} \\ & A_1 \;[\!]\; A_2 & \text{(nondeterministic choice)} \\ & A_1;\; A_2 & \text{(sequential composition)} \\ & A_1 \;/\!/\; A_2 & \text{(prioritised composition)} \end{array}$$

where R is a predicate, x is a variable or a list of variables, and A_1 and A_2 are actions. Semantically, an action A is defined by the *weakest precondition* for A to establish some post-condition Q, denoted $wp(A, Q)$. In this paper, we regard an action A as being of the form $g \rightarrow S$, where g is the *guard* of the action, given by $gA \,\widehat{=}\, \neg wp(A, false)$, and S is the *action body*. An action is said to be *enabled*, if its guard is $true$, *disabled* otherwise. Actions are considered *atomic*, meaning that whenever one is selected for execution, it will be completed without interference. In this paper, we assume that all statements are *conjunctive monotonic predicate transformers*, that is, $\forall p, q.wp(S, (p \wedge q)) = wp(S, p) \wedge wp(S, q)$.

Additionally, the *quantified composition* of actions is defined by

$$[*i = 0 \ldots n : A_i] \,\widehat{=}\, A_0 * A_1 * \ldots A_n$$

where $*$ can be any of the allowed operators.

3.1 Refinement

Action systems are meant to be designed in a stepwise manner within the *refinement calculus* framework [7]. The *refinement calculus* preserves the correctness of the actions during refinement procedure.

The (atomic) action A is said to be *(correctly) refined* by action C, denoted $A \leq C$, if

$$\forall q.(wp(A, q) \Rightarrow wp(C, q))$$

holds. This is equivalent to the condition

$$\forall p, q.((p \; A \; q) \Rightarrow (p \; C \; q))$$

which means that the *concrete* action C preserves every total correctness property of the *abstract* action A.

In the following paragraphs we introduce some rules based on refinement relations.

Rule 1 – Data refinement. Assume two actions A and C with variables a, u and c, u, respectively. Let $R(a, c)$ be a boolean relation between the variables a and c, and $I(c, u)$ an invariant over the action C on the variables c and u. The the abstract action A is *data-refined* by the concrete action C using the *abstraction relation* $R(a, c) \wedge I(c, u)$, denoted $A \leq_{R,I} C$, if

$$\forall q.(R \wedge I \wedge wp(A, q) \Rightarrow wp(C, \exists a.R \wedge I \wedge q) \qquad (1)$$

holds. The predicate $\exists a.R \wedge I \wedge q$ is a boolean condition on the program variables a and c.

Rule 2 – Prioritised composition. The prioritised composition defined in [8] offers us the possibility to concisely impose precedence of some actions / action systems over others.

Briefly, the prioritised composition of two actions is expressed in terms of the non-deterministic choice as: $A \;/\!/\; B \,\widehat{=}\, A \;[\!]\; \neg gA \rightarrow B$. The result of interest to us is the fact that always, a nonprioritised composition $A \;[\!]\; B$ can be turned into a prioritised one by merely strengthening the guard of the less important action. Hence, always:

$$A \;[\!]\; B \leq A \;/\!/\; B \qquad (2)$$

Rule 3 – Distributivity of sequence over the choice. As we confined ourselves to work with conjunctive, monotonic predicate transformers, we may benefit, in our reasoning of the following rule, describing the Distributivity of the sequence over the choice operator [7]:

$$(A \;[\!]\; B); C = (A; C) \;[\!]\; (B; C) \qquad (3)$$

There are several other rules that we will consider in the forthcoming sections, such as the introduction of a local variable or introduction of the removal of an empty statement, for which we do not offer detailed exemplifications. They may be found elsewhere [7, 9].

Execution of an Action System. Starting with the original paper by Back and Kurki-Suonio [4], the sequential execution model was established as a *de facto* reasoning environment for AS designs. Parallel executions are modelled by interleaving actions that have no read / write conflicts.

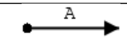
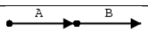
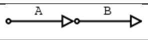
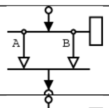
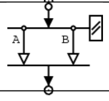
Thus, the execution of an action system assumes that the system is observed by a virtual external entity - the **execution controller** (**controller** in short) - which, at any moment knows what actions, in which action systems, are enabled. Nondeterministically, it selects one of them for execution.

The initialisation places the systems in a stable, starting state. The controller then selects any of the enabled actions for execution, after which the system moves to a new state. We call this operation an *execution round*. Notice that an execution round is equivalent to the execution of an action. After this, the controller evaluates the new state, observes the enabled actions and starts another execution round.

UML Profile for Action Systems An UML profile for Action Systems is a graphical design environment for Action Systems. It is targeted to facilitate the designer's burden by illustrating the system under development. With this graphical representation the designer can compose the system and to manage large complex systems more easily. The operators connecting the actions are clearly visible in the graphical representation and thus the overall functionality becomes apparent. In the table 1 is shown part of the graphical notation defined in [10].

In the notation labels of the arrows correspond to actions in the textual representation. In the prioritised composition the highest priority action goes with the farthermost arrow from the operator symbol. The execution of the action system starts from the arrow with a hollow start and terminates after the end is reached after which a new round is started. The end is marked with a dot inside a circle.

Table 1: Notation of actions and action sequences

Name	Notation	Meaning
Atomic Action		A
Action Sequence		$A; B$
Non-Atomic Action Sequence		A**;** B
Non-Deterministic Composition		$A \mathbin{[\!]} B$
Prioritised Composition		$A \mathbin{/\!/} B$

4 Derivations

We start our design from a single segment bus, where we have the descriptions of masters, the slaves and the bus arbiter. The transformation towards a segmented bus is transparent except for the arbiter, which becomes a *segment arbiter*. The SA has to take into consideration now a higher priority master, represented by the CA. It will also have to consider idling for the period of time when the segment is just a transmission line between a winning master and its selected slave, both situated outside the boundaries of the specific SA segment.

In this section we concentrate on the granting activity only. The notations T and F stand for the boolean values of $true$ and $false$, respectively.

The segment arbiter. The operation of the arbiter on a single bus system consists of two jobs. One is to grant the requesting masters the access to the bus, whenever the previous owner finished the transfers. This is signalled by raising the line *gr* ($gr := T$). The second one is to supervise the current owner so that the number of transfers does not exceed a limit. When this limit is reached, the master is informed that it has lost the control. Hence, the variable *gr* is reset ($gr := F$). With the help of the boolean variable ack the arbiter also informs the masters that the decision on the next owner of the bus is taken. If requesting masters did not receive access to the bus, they are supposed to keep on requesting.

While a granted master is transferring on the bus, the arbiter must not grant any other master. Hence, the supervision activity must have a higher priority than granting.

4.1 The $Grant$ Action

We start from a single master system. Replicating the action in a proper manner will eventually describe the granting activity for a larger number of masters. Hence, our derivation begins with the $Grant$ action given below.

$$\begin{aligned} Grant \quad \widehat{=} \quad & (\, req \wedge gr = F \rightarrow (gr := T \mathbin{[\!]} skip); ack := T) \\ & \mathbf{;}\, \neg req \wedge ack \rightarrow ack := F \end{aligned}$$

The above description specifies that, whenever a master requests the bus (req) and it does have it ($gr = F$), then the arbiter may give access to resources ($gr := T$) or not ($skip$). After this, the ack line is set to $true$. It is set back to $false$ whenever the master also resets the req signal. Following this, a new granting cycle may start.

We apply Rule 3 to the *Grant* action above, and we obtain:

$$\begin{aligned} Grant_1 \quad \widehat{=} \quad & (\, req \wedge gr = F \rightarrow (gr := T; ack := T \mathbin{[\!]} skip; ack := T)) \\ & \mathbf{;}\, \neg req \wedge ack \rightarrow ack := F \end{aligned}$$

Next, consider the relation

$$\begin{aligned} R(gr, grant) \quad \widehat{=} \quad & (gr = F \Leftrightarrow (grant = F \vee grant = Hold)) \wedge \\ & (gr = T \Leftrightarrow grant = T) \end{aligned}$$

The relation R specifies how we can replace the original variable *gr* with the new variable $grant$. From a two-valued grant signal, we move now to a three-valued one. This is required because, when a local master requests an external segment access, the SA cannot grant it without asking the CA. Hence, it will first place the corresponding grant line to a new value, *Hold* and will forward the request to the CA. When the CA grants the access, it informs the corresponding SA, which now forwards the grant to the specific master. Using R, we data refine the granting action ($Grant_1 \leq_R Grant^1$) to

$$\begin{aligned} Grant^1 \quad \widehat{=} \quad & (\, req \wedge (grant = F \vee grant = Hold) \rightarrow grant := T; \\ & ack := T \mathbin{[\!]} req \wedge grant = F \rightarrow grant := Hold; \\ & ack := T\,)\mathbf{;}\, \neg req \wedge ack \rightarrow ack := F \end{aligned}$$

We continue by introducing the variables req_C and ack_C, which implement the communication with the CA. The first one is updated by the SA, while the second is only read by the SA and written by the CA. Hence, we have the description:

$$\begin{aligned} Grant^2 \quad \widehat{=} \quad & (\, req \wedge (grant = F \vee (grant = Hold \wedge ack_C)) \rightarrow \\ & grant := T; ack := T \mathbin{[\!]} req \wedge grant = F \rightarrow grant := \\ & Hold; ack := T; req_C := T\,)\mathbf{;}\, \neg req \wedge ack \rightarrow ack := F \end{aligned}$$

The arbiter differentiates a local request from an external one by reading the slave address lines ($SAddr$) provided by the requesting master. The own address ($lsegnr$) is coded inside the SA. For simplicity, we denote $local \;\widehat{=}\; SAddr = lsegnr$. The *Grant* action is then refined by the following:

$$\begin{aligned} Grant^3 \quad \widehat{=} \quad & (\, req \wedge ((local \wedge grant = F) \vee (grant = Hold \wedge ack_C)) \\ & \rightarrow grant := T; ack := T \mathbin{[\!]} req \wedge \neg local \wedge grant = F \\ & \rightarrow grant := Hold; ack := T; req_C := T\,)\mathbf{;}\, \neg req \wedge ack \rightarrow \\ & ack := F \end{aligned}$$

A master that needs to transfer data to / from a slave placed in an external segment will wait considerably more than a master requesting a local resource. In order to balance this aspect, we decide to give higher priorities to such requests. In the same step, we apply *Rule 3*. We have:

$$\begin{aligned} Grant^4 \quad \widehat{=} \quad & \Big((\, req \wedge grant = Hold \wedge ack_C \rightarrow grant := T; ack := T \\ & \mathbin{[\!]} req \wedge \neg local \wedge grant = F \rightarrow grant := Hold; req_C := \\ & T; ack := T\,)\mathbf{;}\, \neg req \wedge ack \rightarrow ack := F \Big) \mathbin{/\!/} \Big(req \wedge local \\ & \wedge grant = F \rightarrow grant := T; ack := T\mathbf{;}\, \neg req \wedge ack \rightarrow \\ & ack := F \Big) \end{aligned}$$

Here is the point where we take into consideration the existence of several masters within the segment. They are numbered from 0 to nom. Each master j communicates with the SA by means of a request signal $req[j]$, and slave address lines $SAddr[j]$. The SA updates for each master a grant signal, $grant[j]$. masters has also access to the unique ack line used by the SA to signal termination of a granting session. Thus, we replicate the granting activity following a data refinement step which relates the initial variables req and $grant$ to their "vectorised" versions; $req[0 \ldots nom]$ and $grant[0 \ldots nom]$. The abstraction relation is:

$$\begin{aligned} R_1 \quad \widehat{=} \quad & (req = T \Leftrightarrow \exists j \in \{0, \ldots, nom\}.req[j] = T) \\ & \wedge (req = F \Leftrightarrow \forall j \in \{0, \ldots, nom\}.req[j] = F) \end{aligned}$$

and we have $Grant^4 \ \leq_{R_1} \ Grant^5$, where:

$$\begin{aligned} Grant^5 \quad \widehat{=} \quad & \Big(\Big(\Big[[\!]\, j := 0 \ldots \mathit{nom} : req[j] \wedge grant[j] = Hold \wedge ack_C \\ & \rightarrow grant[j] := T; ack := T \Big] \ [\!] \ \Big[[\!]\, j := 0 \ldots \mathit{nom} : req[j] \\ & \wedge \neg local \wedge grant[j] = F \rightarrow grant[j] := Hold; ack := T; \\ & req_C := T \Big] \Big) \mathbf{;} \neg req[j] \wedge ack \rightarrow ack := F \Big) \ /\!/ \ \Big(\Big[[\!]\, j := \\ & 0 \ldots \mathit{nom} : req[j] \wedge local \wedge grant[j] = F \rightarrow grant[j] := T; \\ & ack := T \mathbf{;} \neg req[j] \wedge ack \rightarrow ack := F \Big] \Big) \end{aligned}$$

Observe that there is one situation which is not yet exactly reflected in the above description. It corresponds to the request coming from the CA to the SA, asking access rights for another segment master either to pass through the segment or to access a local resource. The channel devoted to this communication is similar with the ones considered by $Grant^5$: the CA is just another local master from the point of view of the SA. However, we just want to separately identify this specific master, as we intend to place it higher in the priority list. We identify it as the "master[0]". We assign to this master the highest priority. Therefore, we have $Grant^6$:

$$\begin{aligned} Grant^6 \quad \widehat{=} \quad & \Big(\Big(req[0] \wedge grant[0] = Hold \wedge ack_C \rightarrow grant[0] := T; \\ & ack := T \ /\!/ \ \Big[[\!]\, j := 1 \ldots \mathit{nom} : req[j] \wedge grant[j] = Hold \\ & \wedge ack_C \rightarrow grant[j] := T; ack := T \Big] \ [\!] \ req[0] \wedge \neg local \\ & \wedge grant[0] = F \rightarrow grant[j] := Hold; ack := T; req_C := T \\ & /\!/ \Big[[\!]\, j := 1 \ldots \mathit{nom} : req[j] \wedge \neg local \wedge grant[j] = F \rightarrow \\ & grant[j] := Hold; ack := T; req_C := T \Big] \Big) \mathbf{;} \neg req[j] \wedge ack \\ & \rightarrow ack := F \Big) \ /\!/ \ \Big(req[0] \wedge local \wedge grant[0] = F \rightarrow \\ & grant[0] := T; ack := T \ /\!/ \ \Big[[\!]\, j := 1 \ldots \mathit{nom} : req[j] \wedge \\ & local \wedge grant[j] = F \rightarrow grant[j] := T; ack := T \\ & \mathbf{;} \neg req[j] \wedge ack \rightarrow ack := F \Big] \Big) \end{aligned}$$

The last step in the derivation of SA is related to the communication channel between the SA and the CA. Notice that the CA does not provide requested segment addresses. Hence, we may assume that it always requests *local* access, therefore, the corresponding *grant* line can not be placed on *Hold*. At the same time, we rename the variables $req[0]$ and $grant[0]$ as req_O and ack_O, respectively. Thus, we have:

$$\begin{aligned} Grant^7 \quad \widehat{=} \quad & \Big(\Big[[\!]\, j := 1 \ldots \mathit{nom} : req[j] \wedge grant[j] = Hold \wedge ack_C \rightarrow \\ & grant[j] := T; ack := T \ [\!] \ req[j] \wedge \neg local \wedge grant[j] = F \\ & \rightarrow grant[j] := Hold; ack := T; req_C := T \mathbf{;} \neg req[j] \wedge ack \\ & \rightarrow ack := F \Big] \Big) \ /\!/ \ \Big(req_O \wedge ack_O = F \rightarrow ack_O := T; ack \\ & := T \mathbf{;} \neg req_O \wedge ack \rightarrow ack := F \ /\!/ \ \Big[[\!]\, j := 1 \ldots \mathit{nom} : req[j] \\ & \wedge local \wedge grant[j] = F \rightarrow grant[j] := T; ack := T \mathbf{;} \neg req[j] \\ & \wedge ack \rightarrow ack := F \Big] \Big) \end{aligned}$$

Next, we give names to the actions composing the *Grant* operation. Thus, we identify:

$$\begin{aligned} LAckF \,\widehat{=}\, & \neg req[j] \wedge ack \rightarrow ack := F \\ LAckF_O \,\widehat{=}\, & \neg req_O \wedge ack \rightarrow ack := F \\ RoAckC \,\widehat{=}\, & req[j] \wedge grant[j] = Hold \wedge ack_C \rightarrow grant[j] := T; ack := T \\ RoGr \,\widehat{=}\, & req[j] \wedge \neg local \wedge grant[j] = F \rightarrow grant[j] := Hold; \\ & ack := T; req_C := T \\ EGr \,\widehat{=}\, & req_O \wedge ack_O = F \rightarrow ack_O := T; ack := T \\ LGr \,\widehat{=}\, & req[j] \wedge local \wedge grant[j] = F \rightarrow grant[j] := T; ack := T \end{aligned}$$

We rewrite the action $Grant^7$ considering the above notations. In the figure 3 is shown the graphical notation of the $Grant^7$ using the UML profile for Action Systems.

$$\begin{aligned} Grant^7 \quad = \quad & \Big[[\!]\, j := 1 \ldots \mathit{nom} : (RoAckC \ [\!] \ RoGr) \mathbf{;} LAckF \Big] \\ & /\!/ \quad EGr \mathbf{;} LAckF_O \ /\!/ \ \Big[[\!]\, j := 1 \ldots \mathit{nom} : LGr \mathbf{;} LAckF \Big] \end{aligned}$$

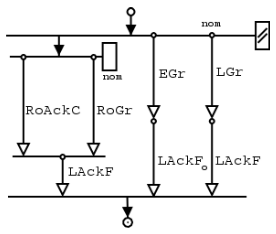


Figure 3: The graphical representation of the $Grant^7$.

5 Conclusions

In this study we have shown how the action systems formalism is applied to correctly derive a segment arbiter specification from a single segment arbiter. The work is part of a larger project that analyses the realisation of a segmented bus, starting from high levels of abstraction down to implementation. The project also examines how a Unified Modelling Language (UML) can be firmly affiliated with the Action Systems. And to adapt the techniques of the action systems framework to the environment provided by the Object Constraint Language (OCL).

References

[1] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall International, 1976.

[2] D. M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems.* PhD thesis, Standford University, 1984.

[3] T. Seceleanu, J. Plosila, P. Liljeberg. *On-Chip Segmented Bus: A Self Timed Approach.* In Proceedings of the 15th IEEE ASIC/SOC Conference, 2002, pages 216-221.

[4] R. J. R. Back and R. Kurki-Suonio. Distributed Cooperation with Action Systems. *In ACM Transactions on Programming Languages and Systems,* Vol. 10, No. 4.1988, pp. 513-554.

[5] J. Plosila. *Self-Timed Circuit Design - The Action Systems Approach.* Ph.D. Thesis, University of Turku, Turku, Finland, 1999.

[6] T. Seceleanu. *Systematic Design of Synchronous Digital Circuits.* Ph.D. Thesis, Abo Akademi, Turku, Finland, 2001.

[7] R. J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Springer–Verlag, 1998.

[8] E. Sekerinski, K. Sere. *A Theory of Prioritizing Composition.* The Computer Journal, VOL. 39, No 8, pp. 701-712.

[9] K. Sere. *Stepwise Derivation of Parallel Algorithms.* Ph.D. Thesis, Abo Akademi, Turku, Finland, 1990.

[10] T.Westerlund, T. Seceleanu. UML Profile for Action Systems. To appear as a TUCS technical report, 2003.