Tomi Metsälä | Tomi Westerlund |
Seppo Virtanen | Juha Plosila

ActionC: An Action Systems Approach to System Design with SystemC

Turku Centre *for* Computer Science

TUCS Technical Report
No 865, January 2008

ActionC: An Action Systems Approach to System Design with SystemC

Tomi Metsälä
University of Turku, Department of Information Technology
Joukahaisenkatu 3-5 B, 20520 Turku, Finland
`tomi.metsala@utu.fi`

Tomi Westerlund
University of Turku, Department of Information Technology
Joukahaisenkatu 3-5 B, 20520 Turku, Finland
`tomi.westerlund@utu.fi`

Seppo Virtanen
University of Turku, Department of Information Technology
Joukahaisenkatu 3-5 B, 20520 Turku, Finland
`seppo.virtanen@utu.fi`

Juha Plosila
University of Turku, Department of Information Technology
Joukahaisenkatu 3-5 B, 20520 Turku, Finland
`juha.plosila@utu.fi`

TUCS Technical Report

No 865, January 2008

Abstract

ActionC is a new approach to rigorous modelling and development of computer systems. ActionC integrates SystemC, an informal design language, and Action Systems, a formal modelling language that supports verification and stepwise correctness-preserving refinement of system models. The ActionC approach combines the possibility to use a formal correct-by-construct method and an industry standard design language with simulation and synthesis support. In our approach Action Systems provides a formal foundation for an informal SystemC model with a promise of verified simulation, refinement and synthesis. At this point we have explored the first aspects of ActionC development: the SystemC implementations of nondeterminism and of Action Systems type inter-module communication. The early experiments have successfully produced simulatable SystemC descriptions of Action Systems.

Keywords: SystemC, Action Systems, Formal methods

TUCS Laboratory
Distributed Systems Design

1 Introduction

Formal methods provide an environment to specify, design and verify systems with the benefits of a strict mathematical basis. In our case formal methods are provided by the Action Systems formalism [1], which offers a powerful stepwise development environment for designing embedded HW/SW systems throughout the design project from abstract specification down to implementable model. Action Systems enables us to formally verify each derivation step within specific refinement calculus framework [2].

Programming languages that are targeted to software development do not natively support hardware-oriented functionality such as clocks, signals, reactivity and parallel processing accurately. Because of this, when such languages and environments are used in embedded system development, the support for hardware modelling is typically built on top of the underlying programming language. The benefit of this approach is that while support for hardware modelling is added, all features of the base language can still be used in system design. One such a language is SystemC [3][4], which is a system modelling and simulation environment based on the standard C++ programming language[5]. In addition to hardware modelling related data types, structures and class definitions, SystemC also provides the system designer all the object oriented techniques available in C++. SystemC is a single language framework for co-verifying systems at multiple, possibly mixed, abstraction levels allowing the system designer to gradually develop the model towards lower abstraction levels without a need to translate the model into a hardware design language.

This report explores the possibilities of the Action Systems formalism in providing a formal foundation for SystemC modelling. The objective is to capitalise the best features of both methods and to combine the results as a new system modelling framework. The proposed *ActionC* framework would provide a SystemC model out of an Action Systems description at a given abstraction level. The utilisation of the ActionC framework would allow the designer to initially produce a simulatable description from a formal Action Systems description. The purpose of ActionC would also be to ensure the preservation of the formal correctness throughout the model's refinement process as well as during the simulation. In this report two important aspects of Action Systems are considered in detail: nondeterminism and communication between system modules. The final conclusion should tell us how SystemC handles these aspects in a simulation and how the ActionC framework development should be continued in the future.

First, the Action Systems formalism is introduced in Sect. 2 with a short description of the language and an introduction to its inter-module communication methods. Section 3 briefly introduces SystemC by presenting its basic structures and their usage. Based on the information presented in the first two sections, Action Systems and SystemC are viewed together in Sect. 4. The matching constructs between the two modelling languages are studied as the foundation for the new ActionC framework. In addition, the first methods for the ActionC class library are introduced in Sect. 4. Section 5 presents an example in which an Action Systems model is transformed into an implementation that follows the coding style of the proposed ActionC framework and utilises the ActionC methods and constructs introduced in the previous section. Concluding remarks are provided in Sect. 6.

2 Action Systems Formalism

Action Systems is a design method for modelling and refining formal models of sequential programs as well as more complex parallel and reactive systems [1][6]. The Action Systems formalism was initially proposed by Ralph-Johan Back and Reino Kurki-Suonio [7] and it is based on *the guarded command language* by Edsger W. Dijkstra [8]. Action Systems allows a system programmer to design a system based on its logical behaviour. The decisions concerning the actual implementation and the questions of sequential and parallel executions in the system can be made after the design of the logical behaviour.

An action system is a program in which the system execution is described in terms of *atomic actions*. Atomic actions, once chosen for execution, are executed to completion without interference from other actions in the system. Only the initial and final states of an atomic action are observable, which means that there are no observable states between them. If two actions do not have any shared variables, it is possible to execute them in parallel. Both parallel and sequential executions of such actions are guaranteed to produce identical results.

The Action Systems formalism supports modularity, including modularisation mechanisms such as procedures, parallel composition and data encapsulation. An action system module has its own local variables and a single iteration statement. Modularity of Action Systems includes also parameterised procedures and nested action systems. The action system modules communicate with each other using different kinds of communication mechanisms. Modules can share variables, which they use to pass information. The information passing can also be accomplished via shared actions and remote procedure calls by importing and exporting variables and procedures within the system module interface.

A stepwise refinement method is used for the development of an actual parallel, distributed system from an initial action system model. The model is produced using a formal method called *refinement calculus*, which was originally proposed by Ralph-Johan Back [2][9]. The refinement is performed for the Action Systems model created as a result of the behavioural and logical design of a system. By following the refinement calculus the refinement steps preserve the total correctness of the original statements. This way the initially general and language independent design is refined into efficiently implementable code in possibly several different programming languages.

2.1 Actions

Actions are defined (for example) by:

$A ::= \ abort$	$($*abortion, non-termination*$)$
$\mid skip$	$($*empty statement*$)$
$\mid x := e$	$($*(multiple) assignment*$)$
$\mid$ **do** A **od**	$($*iterative composition*$)$
$\mid p \rightarrow A$	$($*guarded command/action*$)$
$\mid A_0; \ldots ; A_n$	$($*sequential composition*$)$
$\mid A_0 \ [] \ \ldots \ [] \ A_n$	$($*nondeterministic choice*$)$
$\mid A_0 \ /\!/ \ \ldots \ /\!/ \ A_n$	$($*prioritised composition*$)$
$\mid A_0 * \ldots * A_n$	$($*simultaneous composition*$)$
$\mid \{p\}$	$($*assertion statement*$)$
$\mid [p]$	$($*assumption statement*$)$
$\mid x := x'.R$	$($*nondeterministic assignment*$)$
$\mid \mid[\mathbf{var}\ x := x_0;\ A]\mid$	$($*block with local variables*$)$

where A, A_0 and A_n, $n \in \mathbb{N}^+$, are actions; x is a variable or a list of variables; x_o some value(s) of variable(s) x; e is an expression or a list of expressions; and p and R are Boolean conditions. The *total correctness* of an action A with respect to a precondition P and a postcondition Q is denoted PAQ and defined by:

$$PAQ \widehat{=} P \Rightarrow \mathbf{wp}(A, Q)$$

where $\mathbf{wp}(A, Q)$ stands for the *weakest precondition* for the action A to establish the postcondition Q. The activation of the statement list A is guaranteed to lead to a properly terminating activity leaving the system in a final state that satisfies the postcondition Q and also the weakest precondition.

The guard gA of an action A is defined by:

$$gA \ \widehat{=} \ \neg\mathbf{wp}(A, false)$$

In the case of a guarded action $A \ \widehat{=} \ p \rightarrow B$, we have that $gA = p \wedge gB$. An action A is said to be *enabled* in states, where its guard is true and *disabled*, where the guard is false. Now for the above introduced actions we can define:

$$
\begin{aligned}
\mathbf{wp}(abort, Q) &= false \\
\mathbf{wp}(skip, Q) &= Q \\
\mathbf{wp}(x := e, Q) &= Q[e/x] \\
\mathbf{wp}(\mathbf{do}\ A\ \mathbf{od}, Q) &= (\exists k.\ k \geq 0 \wedge H(k)) \qquad where \\
&\quad k = 0 \Rightarrow H(0) = Q \wedge \neg gA \\
&\quad k > 0 \Rightarrow H(k) = (gA \wedge \mathbf{wp}(A, H(k-1))) \vee H(0) \\
\mathbf{wp}(p \rightarrow A, Q) &= P \Rightarrow \mathbf{wp}(A, Q) \\
\mathbf{wp}((A_0; A_1), Q) &= \mathbf{wp}(A_0, \mathbf{wp}(A_1, Q)) \\
\mathbf{wp}((A_0\ [\!]\ A_1), Q) &= \mathbf{wp}(A_0, Q) \wedge \mathbf{wp}(A_1, Q) \\
\mathbf{wp}(A_0 /\!/ A_1, Q) &= (\mathbf{wp}(A_0, Q)) \wedge (\neg gA_0 \Rightarrow (\mathbf{wp}(A_1, Q))) \\
\mathbf{wp}(A_0 * A_1, Q) &= gA_0 \wedge gA_1 \Rightarrow (\forall a_0', a_1'.P_{A_0} \wedge P_{A_1} \Rightarrow Q[a_0', a_1'/a_0, a_1]) \\
\mathbf{wp}(\{p\}, Q) &= p \wedge Q \\
\mathbf{wp}([p], Q) &= p \Rightarrow Q \\
\mathbf{wp}(x := x'.p, Q) &= \forall x'.p \Rightarrow Q[x'/x] \\
\mathbf{wp}(|[\mathbf{var} x; A]|, Q) &= \forall x.(\mathbf{wp}(A, Q))
\end{aligned}
$$

The above defined actions and their compositions are all atomic actions. *Atomic compositions* are larger atomic entities composed of simpler ones, and the actions within such compositions are called *merged* actions. However, in a *non-atomic composition* of actions the component actions are atomic entities of their own, but the composition itself is not. One such a construct is the iterative composition, the **do-od** loop, whose execution may consist of several executions of its component actions. Non-atomicity means that also the intermediate states of the composition can be observed in contrast to an atomic composition.

2.2 Action System

An action system $\mathcal{M}$ has the form:

```
sys $\mathcal{M}$ (imp $p_I$; exp $p_E$; )( $g$; )
|[  private procedure
      $p(\textbf{in}\ x; \textbf{out}\ y) : (P)$;
    public procedure
      $p_E(\textbf{in}\ x; \textbf{out}\ y) : (P_E)$;
    variable
      $l$;
    action
      $A_i : (aA_i)$;
    initialisation
      $g, l := g0, l0$;
    execution
      do composition of actions $A_i$ od    ]|
```

where we can identify three main sections: *interface*, *declaration* and *iteration*. The interface part declares the global variables g, which are visible outside the action system boundaries meaning that they are accessible by other action systems. These variables may be either **in**, **out** or **inout** variables. The interface also introduces *interface procedures* p_I and p_E that are imported or defined and exported by the system, respectively. In general, procedures are any atomic actions A, possibly with some local variables w that are initialised to $w0$ every time the procedure is called. The action A can access the global (g) and local (l) variables of the host/enclosing system and the formal parameters x and y. Procedures can be treated as parametrisable subactions because their executions are considered as parts of the calling action. An action system that does not have any interface variables or procedures is *a closed action system*. Otherwise it is *an open action system*. The declarations part introduces all the local variables l, local procedures p, exported procedures p_E and actions A_i that perform operations on local and global variables.

The operation of the action system is started by the initialisation in which the variables are set to their predefined values. In the iteration part, in the **execution** section, actions are selected for execution based on their composition and enabledness. This is continued until there are no enabled actions, whereupon the computation terminates. Hence, an action system is essentially an initialised block with a body that contains a repeatedly executing statement.

2.3 Procedure Based Communication

The procedure based communication [10] uses remote procedures to model communication channels between action systems. One action system defines and exports a procedure, which is imported by another system. This gives the importing system a possibility to perform a remote call by executing the procedure (Fig. 1). Consider the action systems $\mathcal{S}nd$ and $\mathcal{R}ec$ whose internal activities are denoted

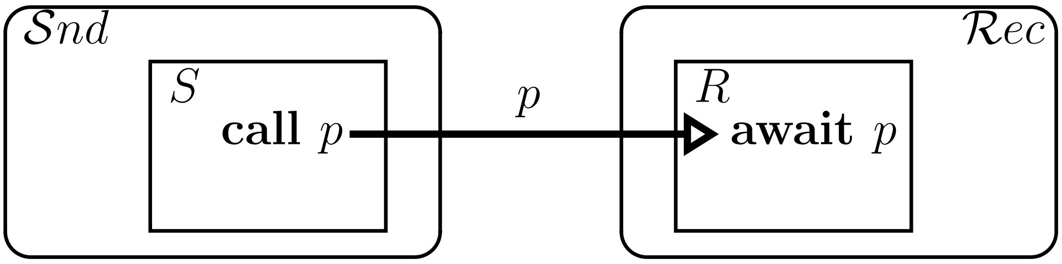

Figure 1: Action systems $\mathcal{S}nd$ and $\mathcal{R}ec$ communicating directly with each other using a procedure

with actions

$$S \mathrel{\widehat{=}} (S_1; \textbf{call } p(l_{snd}); S_2)$$

and

$$R \mathrel{\widehat{=}} (R_1; \textbf{await } p; R_2)$$

where $S_{1,2}$, $R_{1,2}$ are arbitrary subactions of S and R, respectively. Interface procedure p is defined in and exported by the receiver $\mathcal{R}ec$, and imported and called by the sender $\mathcal{S}nd$, with the sender's local variables l_snd as actual value parameters. Furthermore, the local variables of the systems are distinct, $l_{snd} \cap l_{rec} = \emptyset$, communication variables are a set $g_{snd} \cap g_{rec}$ and the initialisations of the communication variables $g_{snd} \cap g_{rec}$ are consistent with each other. The body P of p can be any atomic action writing onto the receiver's local variables l_{rec}. In *parallel composition* of action systems, $\mathcal{S}nd \parallel \mathcal{R}ec$, the **execution** clause of the composed system is by definition:

$$\textbf{do } S \;[\!]\; R \textbf{ od}$$

The construct $S \;[\!]\; R$, where S calls p (**call** command) and R awaits such a call (**await** command), is regarded as a single atomic action SR, defined by:

$$SR \mathrel{\widehat{=}} (S_1; R_1; P[l_{snd}/x]; R_2; S_2)$$

Therefore, communication is based on sharing an action in which data is atomically passed from $\mathcal{S}nd$ to $\mathcal{R}ec$ by executing the body P of the procedure p while hiding the communication details into the procedure call.

3 SystemC

In the same way as the Action Systems formalism, SystemC [3][4] can be seen as a methodology that can be used in writing a specification for a system that includes both hardware and software components. SystemC is a class library for the standard C++ programming language [5], and it is an open source standard, which is currently supported and advanced by the Open SystemC Initiative (OSCI)

[11]. IEEE has approved the IEEE 1666 standard for version 2.1 of the OSCI SystemC Language Reference Manual [12], which is also in this report the main reference concerning the SystemC class library definitions.

With SystemC a system designer can create cycle-accurate models of software algorithms, hardware architectures and interfaces of SoC and system-level designs. SystemC is a fine tool for hardware modelling, even though it is built on a high-level software programming language. Therefore, in a HW/SW system design process SystemC can be used inplace of actual hardware description languages, such as VHDL and Verilog. SystemC provides system architecture constructs, such as hardware timing, concurrency, and reactive behaviour, which are not included in standard C++. In addition, the designer can use all the object-oriented features and development tools of C++. Both software and hardware partitions of a system model can be written in a single high-level language, which provides higher productivity, less code and decreases the possibility of errors. Because both software and hardware partitions are written in SystemC, they can be tested using the same test bench without any need for language conversions between different abstraction levels.

3.1 A SystemC Model

A SystemC *module* is the building block of a SystemC model. Breaking a design model into several small pieces makes the otherwise complex system easier to manage. Modules implement data encapsulation by hiding local data and algorithms from other modules in the system. A module may contain a hierarchy of other modules. Modules run *processes* that are triggered by *events*. Processes describe the behaviour of the modules, to which they are confined. Modules are connected to each other by *channels*, which are used in inter-module and inter-process communication.

SystemC module uses *ports* and *exports* to access channels. SystemC provides three different kinds of ports to allow single-direction access from the outside environment to the module, from the module to the environment or bidirectional access through one port. A port communicates with its designated channel through an *interface*, which gives the port the methods that it can use to access the channel. This way the port acts as an intermediary for the module, while the interface provides the same service for the channel. There can be two types of channels: *primitive channels* and *hierarchical channels*. A primitive channel, sometimes also called a signal, is considered atomic because it does not contain any other SystemC structures. Hierarchical channels may contain other modules and channels as well as internal processes, therefore in practise, they are as complex as modules. Inter-module communication can be refined by using *adapters*, *wrappers* and *converters* if the ports and interfaces between the modules do not match.

A SystemC *simulation* is set up in the *elaboration* phase in which the top level modules, channels and clocks are instantiated and module ports are bound to the channel instances. Constructs inside the inner hierarchies of modules and hierarchical channels are instantiated in their constructors. This way the elaboration process advances recursively from top and down through the entire hierarchy. In the simulation phase the SystemC *scheduler* acts as a system kernel that handles the timing and order of the process execution. It controls event notifications and updates channels when requested.

4 ActionC

In the context of ActionC, Action Systems and SystemC will now be viewed together with the objective to utilise the best parts of both system modelling languages. Action Systems is a formal language that is useful especially at the first stage of a system development process, while SystemC is a powerful tool for system model simulation, refinement, testing and implementation. An initial Action Systems model can be created directly from the system specification and then refined down to the desired abstraction level through correctness preserving refinement steps. By combining the formal features of Action Systems with the benefits of the SystemC environment we could write an executable specification that is based on a formal system description. The formal correctness of this specification would be verified, the specification would be simulatable and, if necessary, synthesisable within the limits set by the synthesisable subset of SystemC.

ActionC can be viewed as the implementation of Action Systems formalism in the SystemC environment. In this report, the key points are the implementations of *nondeterminism* and *communication between action systems*. In Action Systems, these are essential features when modelling complex embedded systems with multiple modules and processors, and therefore must be properly handled also in the ActionC framework. Simultaneous execution of actions is one of the action compositions in Action Systems. In SystemC, simultaneous process execution is implemented with the request-update method and the two-phase semantics of the SystemC scheduler. Therefore, concurrent execution of several actions can be easily simulated in SystemC without any greater need for additional ActionC features. By ActionC features, we mean C++ or SystemC methods or classes that bring the principles of Action Systems formalism into a SystemC model. Implementations of nondeterminism and inter-module communication are elaborated in the forthcoming sections.

The view on the ActionC modelling framework starts with the identification of the language constructs that the Action Systems formalism shares with the SystemC design language. These directly mappable language constructs are gathered in Table 1. A SystemC module corresponds to an action system, both enclosing a local scope. Both also have an interface that they use in communicating with the environment and other similar elements. When concerning ActionC, let us call this structure an *ActionC module*. Action Systems actions are implemented as

Table 1: Matching language constructs between Action Systems and SystemC

Action Systems		**SystemC**
action system	⇔	Module
action	⇔	Module member
do-od loop	⇔	thread process
in var, **out** var	⇔	sc_in<>, sc_out<>
inout variable	⇔	sc_inout<>
non-atomic sequence **;**	⇔	sequential execution
proc	⇔	C/C++ void method

member functions of the ActionC module. The member functions may be either normal C++ methods, SystemC method processes or, if the member function execution needs to be suspended and reactivated, they are implemented as SystemC thread processes. Let us here call these member functions *ActionC actions*. These ActionC actions are executed by performing *action calls*. The **do-od** loop of each of the Action Systems module is also implemented as a thread process that calls the action processes in the composition one at a time. The thread is suspended when the action procecesses and the processes in other modules are executed. When inter-module communication in Action Systems uses only basic data types, the SystemC primitive channel ports `sc_in<>`, `sc_out<>` and `sc_inout<>` match directly with the **in**, **out** and **inout** variables of the Action Systems formalism. However, if the channel structure is more complex, a SystemC hierarchical channel is a more practical solution. The ActionC implementation of the hierarchical channel and its usage is given in the forthcoming sections.

4.1 Nondeterministic Choice

The challenge in modelling the behaviour of an action system model in SystemC is how to capture the behaviour of a nondeterministic choice $' \, [] \, '$, the basic building block in Action Systems. In a nondeterministic composition there are no guarantees that all the enabled actions will be executed. This means that some actions may be executed multiple times, while others may be left untouched. The **do-od** loop of a system is executed as long as there are enabled actions but *weak fairness* is not applied, that is, all actions have equal probabilities of getting chosen for execution during each loop iteration. This behaviour can be modelled in SystemC with the help of a random number generator. Here we implement the random number generator as `ac_selector()`, a function that chooses one of the enabled nondeterministically combined actions for execution. The information on the enabledness of each of the actions is stored onto a boolean variable array `enabled`, which is examined by the `ac_countEnabled()` function. The function counts the number of the enabled actions and passes the information for the `ac_selector()`.

These functions are used by a SystemC structure that simulates the behaviour of a **do-od** loop. After a function call to `ac_selector()` the structure uses the returned integer value to determine the next action to be executed. With an atomic action the function would be called only once, but in the iteration part the call is repeated as long as there are enabled actions. A decrease in the number of enabled actions also decreases the value passed to the random number generator. Hence, when this value evaluates to zero, execution stops and the operation of the system is terminated.

To exemplify the *ActionC iteration loop* structure, we have actions A and B that are chosen for execution nondeterministically. This Action Systems iteration loop **execution do** $A \mathbin{[\!]} B$ **od** can be implemented with the following structure:

```
int num_actions;       // total number of actions in the composition
int action_to_execute; // indexing variable for the inner loop
int action_number;     // the actual action to be executed
int enabled_left;      // number of enabled actions
bool enabled[2];       // actions are enabled/disabled
bool* action_list;     // pointer to the array enabled[].
...

action_number = 0;
enabled_left = ac_countEnabled(action_list,num_actions);

while (enabled_left > 0) {   /* The do-od loop begins */
  action_to_execute = ac_selector(enabled_left);
  while (action_to_execute > 0) {
    if (enabled[action_number]==true) {
      action_to_execute--;
    }
    action_number++;
  }
  switch (action_number) {
    case 1:
      /* execution of action process A */
      break;
    case 2:
      /* execution of action process B */
      break;
  }
  action_number = 0;
  enabled_left = ac_countEnabled(action_list,num_actions);
}       /* The do-od loop ends */
```

where the inner *while* loop uses the random integer value to choose one of the

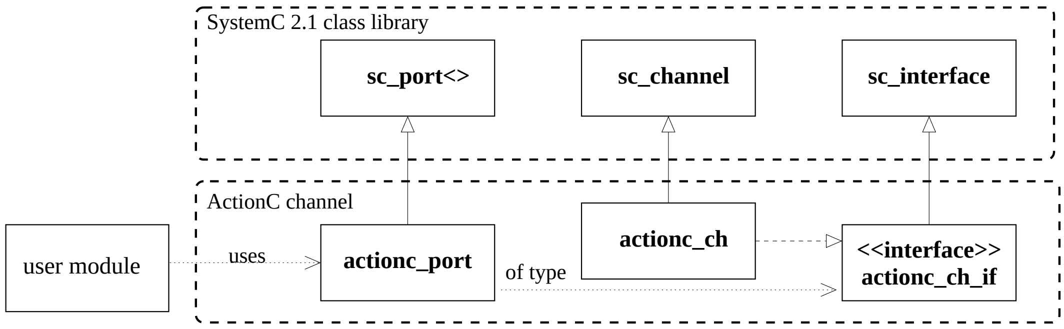

Figure 2: ActionC communication channel class hierarchy

enabled actions, and the outer one keeps the process running as long as there are enabled actions.

4.2 Procedure Based Communication

In Action Systems, the interface variables and procedures form the communication channels between modules. Previously we examined the directly matching language constructs between Action Systems and SystemC and found the SystemC counterparts for the Action Systems interface variables. However, only interface variables of basic data types can be mapped directly to SystemC. When more complex structures are needed, interface variables must be implemented with SystemC hierarchical channels, ports and interfaces as shown in Fig. 2. They compose an *ActionC communication channel,* which confines the functionality of the communication events and hides the actual implementation from the communicating systems. Ports and interfaces are used in connecting a hierarchical channel to the systems so that we have a port-interface couple at each end of the channel. The information passing through the channel is temporarily stored in the local variables of the channel. The communicating systems may access the information in the channel by using the methods declared by their interfaces. Therefore, the access rights to the channel can be controlled by the interface method declarations. Depending on the case, the sets of methods at each end may be identical or different. In the case of **inout** variables, the access rights are equal for both parts but when using **in** and **out** variables, the sets of interface methods differ at each end.

The implementation of an Action Systems interface procedure is a specialisation of the interface variable implementation. Also the procedure based communication can be implemented with hierarchical channels and port-interface couples. In this case, the sets of interface methods at each end are always different because of the difference in their intended functionality. One end represents an exported procedure, while the other end simulates the functionality of an imported procedure. Therefore, for each communication procedure, we have one channel and two port-interface couples: one for an exported and one for an imported procedure. At

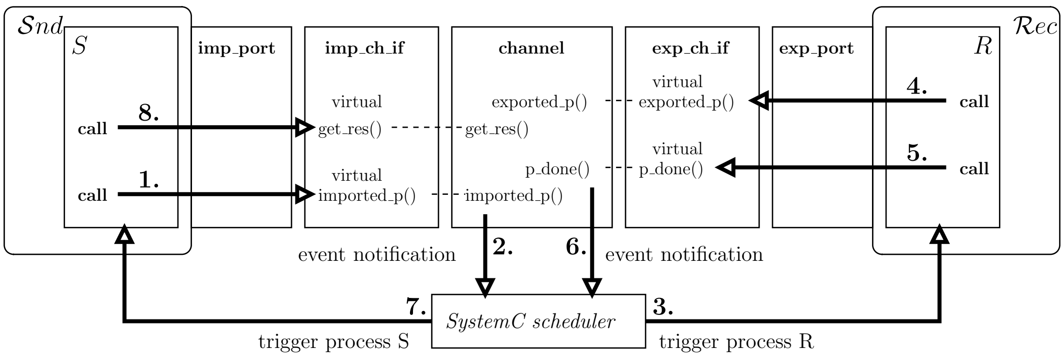

Figure 3: The ActionC implementation of the systems $\mathcal{S}nd$ and $\mathcal{R}ec$ communicating with each other using the procedure based communication model

the imported side, the interface has a minimum amount of methods because the properties of the imported procedure are introduced by the system that exports it. At the exported side, the system has a wide-ranging set of methods to use the channel.

The operation of the ActionC procedure based communication channel, in terms of the Action Systems communication procedure call, is illustrated in Fig. 3. The data integrity during communication call, as required by the Action Systems model, is ensured in the ActionC model by seven (7) or eight (8) communication phases. The number is determined by the existence of a return value. The phases are: (1) The master $\mathcal{S}nd$ starts the communication by calling the channel's interface method as it would call the imported procedure in the Action Systems model. Then the method stores its parameters into the channel's local variables before (2) sending an event notification to the SystemC scheduler that (3) triggers process *R*, which is sensitive to the particular event. The sensitivity of the process *R* corresponds to the function of the **await** command in the Action Systems model. After being invoked (4) the process *R* calls the method(s) provided by its interface to receive and use the parameters stored in the channel's local variables. Process *R* performs its tasks after which it (5) employs an interface method (6) to send an event notification to the SystemC scheduler in order (7) to inform the process *S* of the completion of the communication procedure. (8) If the procedure call produces a return value, the process *S* collects it from the channel using a method provided by its interface.

5 Experimenting with ActionC

In this section we experiment with the previously introduced ActionC features by modelling a system consisting of parallel operating modules. To test the proposed ActionC framework in practise, we introduce an Action Systems model that simulates the basic activities in a simple bank office. The model is then implemented in SystemC with the help of the ActionC methods and coding style. The concentration will be on the implementation of inter-module communication and non-determinism. In the following discussion of the modelling, certain complicated implementational details have been omitted in order to simplify the presentation.

5.1 Action Systems Model

Let us have a model of a bank office whose customers either deposit money to or withdraw money from their bank accounts. Inside the bank office there are two *bank clerks*, A and B, who serve the customers that enter the premises through the *main entrance*. At the entrance, the incoming customer chooses randomly between one of the two clerk to do business with. At the clerk's desk, the customer determines the sum of money that is either deposited to or withdrawn from the customer's account. In order to keep the implementation simple, the sum is chosen at random from integer values between 1 and 1000. The customer makes, again, a random choice between the deposit and withdrawal transactions. The clerk acts according to the customer's wishes and begins to perform the chosen transaction. Before gaining access to the *bank accounts*, the clerk requests a permission from the *arbiter system*, which controls the traffic between the clerks and the bank accounts letting only one clerk at a time access the accounts. After gaining the access, the clerk performs the ordered transaction and releases the access to the bank accounts. The customer has hereby finished his business with the clerk and exits the bank office, while the clerk begins waiting for the next customer to come in through the entrance.

The Action Systems model consists of five parallel operating modules, which correspond to the different scenes in the bank office: $\mathcal{E}ntrance$, $\mathcal{C}lerkA$, $\mathcal{C}lerkB$, $\mathcal{A}rbiter$ and $\mathcal{B}ank$:

$$\mathcal{E}ntrance \parallel \mathcal{C}lerkA \parallel \mathcal{C}lerkB \parallel \mathcal{A}rbiter \parallel \mathcal{B}ank$$

Next we describe the models for each system by using the Action Systems formalism after which we concentrate on their ActionC implementations.

5.1.1 $\mathcal{E}ntrance$

The action system $\mathcal{E}ntrance$ is of form:

```
sys $\mathcal{E}ntrance$ $($out $customerA, customerB\colon Int;$
                    inout $enterA, enterB\colon Bool;$ $)$
                  $[\ ACCOUNTS\colon Int := 5;\ ]$ ::
|[ variable
    $account\colon Int;$
  action
    $NewCustomer\colon (account := c'.(1 \le c' \le ACCOUNTS));$
    $QueueA\colon (\neg enterA \to enterA := T; customerA := account);$
    $QueueB\colon (\neg enterB \to enterB := T; customerB := account);$
  initialisation
    $account, customerA, customerB, enterA, enterB := 0,0,0,0,F,F;$
  execution
    do $NewCustomer;(QueueA\ [\!]\ QueueB)$ od     ]|
```

where the variable *ACCOUNTS* defines the number of accounts the bank manages. Because there is only one account for each customer, *ACCOUNTS* also defines the total number of customers. From that number, action *NewCustomer* nondeterministically selects the next customer that enters the bank office. The clerk the customer chooses to do business with is chosen in the **do-od** loop of the system. Depending on the result of the nondeterministic choice, the customer is directed either to queue A or queue B of clerk A and clerk B, respectively.

5.1.2 $\mathcal{C}lerkA$ and $\mathcal{C}lerkB$

The action system $\mathcal{C}lerkX$ has a form:

```
sys $\mathcal{C}lerkX$ $($imp $Put($in $addr, x\colon Int);$
                 imp $Get($in $addr\colon Int;$ inout $x\colon Int);$
                 in $customerX\colon Int;$
                 inout $enterX\colon Bool;\ comX\colon comchannel;$ $)$
|[ variable
    $action\colon Bool;\ account, sum\colon Int;$
  action
    $Customer\colon (enterX \to sum := sum'.1 \le sum' \le 1000$
        $;\ account := customerX$
        $;\ (action := T\ [\!]\ action := F));$
    $ReqAccess\colon (\neg comX.ack \to comX.req := T);$
    $Deposit\colon (comX.ack \wedge action \to$ call $Put(account, sum);$
    $Withdrawal\colon (comX.ack \wedge \neg action \to$ call $Get(account, sum));$
    $RelAccess\colon (comX.ack \to comX.req := F);$
    $WaitNext\colon (enterX := F);$
  initialisation
    $action, account, sum, enterX, comX := T,0,0,F,(F,F);$
  execution
    do $Customer;ReqAccess;(Deposit\ [\!]\ Withdrawal)$
                  $;RelAccess;WaitNext$ od     ]|
```

where by substituting X for A and B we obtain $\mathcal{C}lerkA$ and $\mathcal{C}lerkB$, respectively. Clerks communicate with their environment, that is, the other action systems, with the imported communication procedures and variables of which the type of the last variable is a record *comchannel*:

$$\textbf{type}\ comchannel : \textbf{record}(req, ack : Bool)$$

The *comchannel* is used in communication with the $\mathcal{A}rbiter$ system. The customer's nondeterministic choice of the money sum involved in the operation as well as the choice between the deposit and withdrawal transactions is performed in the *Customer* action. That is, the *Customer* action models the customer that requests the clerk to perform either the *Deposit* or *Withdrawal* actions. Action *ReqAccess* requests bank access from the arbiter, and *RelAccess* notifies the arbiter after when access is no longer needed. The action systems $\mathcal{C}lerkA$ and $\mathcal{C}lerkB$ communicate with all the other systems in the model excluding themselves. Communication between the clerks and the $\mathcal{B}ank$ system is modelled using the procedure based communication model while the variable based communication is used between the clerks and the $\mathcal{E}ntrance$ and $\mathcal{A}rbiter$ systems.

5.1.3 $\mathcal{A}rbiter$

The action system $\mathcal{A}rbiter$ is of form:

```
sys $\mathcal{A}rbiter$ ($\textbf{inout}$ $comA, comB : comchannel;$ )
|[ action
   $ClerkA: (comA.req \wedge \neg comB.ack \rightarrow comA.ack := T;$
        $[]\ \neg comA.req \wedge comA.ack \rightarrow comA.ack := F);$
   $ClerkB: (comB.req \wedge \neg comA.ack \rightarrow comB.ack := T;$
        $[]\ \neg comB.req \wedge comB.ack \rightarrow comB.ack := F);$
 initialisation
   $comA, comB = (F,F),(F,F);$
 execution
   do $ClerkA\ []\ ClerkB$ od      ]|
```

where actions *ClerkA* and *ClerkB* are chosen for execution based on the Boolean type interface variables $comX.req$ and $comX.ack$, where $X \in \{A, B\}$. If both clerks request access to the bank at the same time, the selection between them is performed in a nondeterministic manner.

5.1.4 $\mathcal{B}ank$

The action system $\mathcal{B}ank$ has a form:

```
sys $\mathcal{B}ank$ $($**exp** $Put($**in** $addr, x\colon Int);$
          **exp** $Get($**in** $addr\colon Int;$ **inout** $x\colon Int);\ )$
        $[\ ACCOUNTS\colon Int := 5;\ ]$ $::$
$|[$ **variable**
    $sdb[ACCOUNTS], account, sum\colon Int;$
  **public procedure**
    $Put($**in** $addr, x\colon Int)\colon (account := addr;\ sum := x$
        $;\ sdb[account-1] := sdb[account-1] + sum);$
    $Get($**in** $addr\colon int;$ **inout** $x\colon int)\colon$
        $(account := addr;\ sum := x$
          $;\ (sum \le sdb[account-1] \rightarrow$
                $sdb[account-1] := sdb[account-1] - sum$
          $[]\ sum > sdb[account-1] \rightarrow$
                $sdb[account-1] := 0));$
  **action**
    $Save\colon ($**await** $Put);$
    $Load\colon ($**await** $Get);$
  **initialisation**
    $sdb[0..ACCOUNTS-1], account, sum := (1000..1000), 0, 0;$
  **execution**
    **do** $Save\ []\ Load$ **od**        $]|$
```

where *sdb*[*ACCOUNTS*] is the safe deposit box array that holds the information on each customer's bank account balance. $\mathcal{B}ank$ exports interface procedures *Put* and *Get*, which are imported and called by action systems $\mathcal{C}lerkA$ and $\mathcal{C}lerkB$. Actions *Save* and *Load* actively wait for procedure calls from $\mathcal{C}lerkA$ and $\mathcal{C}lerkB$ and, when a call comes in, they perform the ordered tasks. If the sum that is requested with the *Get* procedure exceeds the account's balance, only the sum as large as the balance is withdrawn from the account. Negative balance is not allowed.

5.2 ActionC Implementation

The next step is to translate the presented Action Systems model to a corresponding ActionC model using the methods provided by the SystemC class library. The entire implementation of the bank office model is illustrated in Fig. 4, where the `Entrance` module communicates with module `ClerkA` through primitive channels `customerA` and `enterA`, and with module `ClerkB` through primitive channels `customerB` and `enterB`. The arrow at each port determines the direction the corresponding channel transfers the data. ActionC communication channels are illustrated in Fig. 4 as dashed-lined boxes with the type of the channel given inside. Exported procedures *Put* and *Get* are implemented as ActionC constructs that we here name as *put* and *get*, respectively. The *put* and *get* channels con-

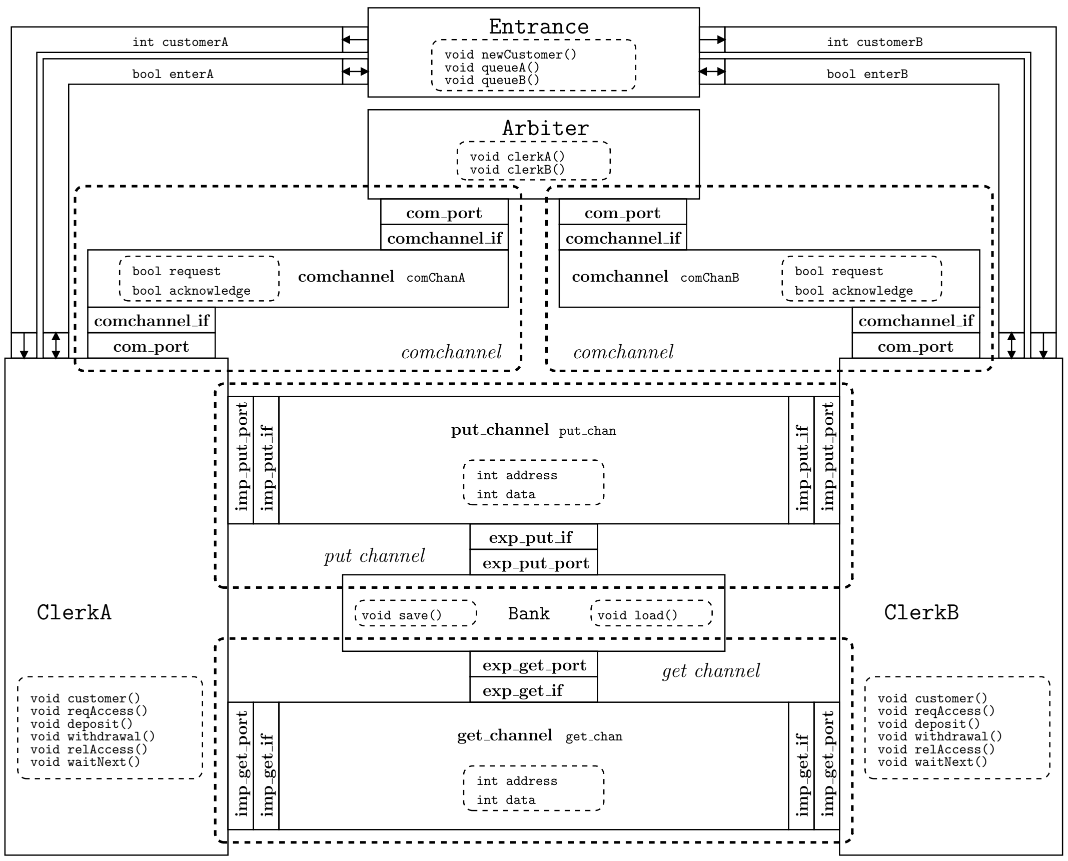

Figure 4: Illustration of the bank office model in ActionC constructs.

sist of hierarchical channels `put_channel` and `get_channel`, respectively, and their related port-interface couples for each channel user. The `Bank` module that exports the procedures, uses the channels through an exported type port-interface couple while the clerks use imported type port-interface couples at their ends. In the communication between the clerks and the `Arbiter` module, the interface variable *comchannel* is implemented as a point-to-point hierarchical channel with equal port-interface couples at its ends. The SystemC code that Fig. 4 illustrates, will be introduced later.

Most of the ActionC implementation is created by directly replacing the Action Systems model structures with the matching SystemC constructs that were presented in Sect. 4. In Action Systems both local and global variables are initialised by the **initialisation** clause, whereas in SystemC this is performed inside the module constructor. Guarded commands are implemented by using the C++ **if** statement in evaluating the state of the guard. The *Int* and *Bool* type interface variables are implemented as SystemC ports of corresponding types bound to `sc_signal` type primitive channels.

5.2.1 Handling ActionC Actions

The communications inside ActionC modules are handled with Boolean signals. Actions that are implemented as SystemC processes are activated by the positive edge of the signal that controls the action's execution, which means that there is one signal for each action process. Most action processes are activated by their host module's *`exec()` process*, which is a SystemC thread that models the behaviour of an Action Systems' iteration part. The `exec()` process is used to handle the action activation in all the modules excluding the `Bank` module whose actions only react to *method calls* from other modules. Method calls are sufficient as the Action Systems actions *Save* and *Load* only include the **await** command. Furthermore, action processes can be triggered directly by the event notifications from the channel that corresponds the Action Systems interface procedure. We illustrate this by the following example. Let us have an Action Systems action *A* that is implemented with a process `actionA()` in SystemC. The execution of the process is handled by an `exec()` process using the following structure:

```
actionA_sig.write(true);
wait(actionA_sig.negedge_event());
```

where the Boolean signal `actionA_sig` is set to true. This activates the process `actionA()` to perform its tasks. After performing its operation, `actionA()` sets `actionA_sig` to false, which, in turn, triggers the `exec()` process. If actions are not implemented as SystemC processes but as standard C++ member functions, the `exec()` process performs a simple method call and waits for return. In this example, however, only processes were used. In the following sections, we will elaborate the implementation of nondeterminism and procedure based communication.

5.2.2 Nondeterminism

Let us first examine the source code of the processes `exec()` and `customer()` of the `Clerk` module. A part of the file `clerk.cpp` is listed below:

```
/* Iteration loop handling actions Customer, ReqAccess, Deposit,
   Withdrawal, RelAccess and WaitNext */
void Clerk::exec() {
  while(true) {
    // activate action Customer
    customer_sig.write(true); wait(customer_sig.negedge_event());

    // activate action ReqAccess
    reqAccess_sig.write(true); wait(reqAccess_sig.negedge_event());

    // nondeterministic choice between Deposit and Withdrawal
    if (action) {
      // activate action Deposit
      deposit_sig.write(true); wait(deposit_sig.negedge_event());
    }
    else {
      // activate action Withdrawal
      withdrawal_sig.write(true); wait(withdrawal_sig.negedge_event());
    }

    // activate action RelAccess
    relAccess_sig.write(true); wait(relAccess_sig.negedge_event());

    // activate action WaitNext
    waitNext_sig.write(true); wait(waitNext_sig.negedge_event());

    wait(enter.posedge_event());
  }
}

/* action Customer */
void Clerk::customer() {
  while(true) {
    sum = ac_selector(1000);        // The sum to use
    account = customer.read();      // Inquiring the account
    int choice = ac_selector(2);    // Choosing action
    if (choice == 1) {
      cout << "Clerk(Customer): Clerk will perform a deposit" << endl;
      action = true;
    }
    else {
      cout << "Clerk(Customer): Clerk will perform a withdrawal" << endl;
      action = false;
    }
    customer_sig.write(false);
    wait();
  }
}
```

The nondeterministic choice between the execution of *Deposit* and *Withdrawal* actions is made by the action *Customer*. The ActionC function `ac_selector()` is used in choosing between the two options. In this case, the `exec()` process does not include the iteration loop structure, because only a simple ***if*** statement is needed to implement the choice between the two integer values. However, the `exec()` processes of `Entrance` and `Arbiter` modules take advantage of the proposed structure, because the nondeterministic choice is performed by the `exec()` process itself, not by a component action. As presented in Table 1, non-atomic sequences in **do-od** loops are implemented as sequential executions.

In the action process `customer()`, the `ac_selector()` function is also used in choosing a random value for the sum that is used in the chosen transaction. This corresponds to the nondeterministic assignment that is used in the Action Systems model. Nondeterministic assignment is also implemented in the module `Entrance`, where the action process `newCustomer()` chooses the customers that enter the bank office.

The action system $\mathcal{A}rbiter$ continuously monitors the states of the **inout** variables *comA* and *comB* of the type *comchannel*. The ActionC implementation of $\mathcal{A}rbiter$ similarly waits for changes in the channel's local variables `request` and `acknowledge`. When a state change has been detected, the iteration loop reacts by executing both of the action processes `clerkA()` and `clerkB()` in a nondeterministic order. The reason for executing both processes is that the `exec()` process can

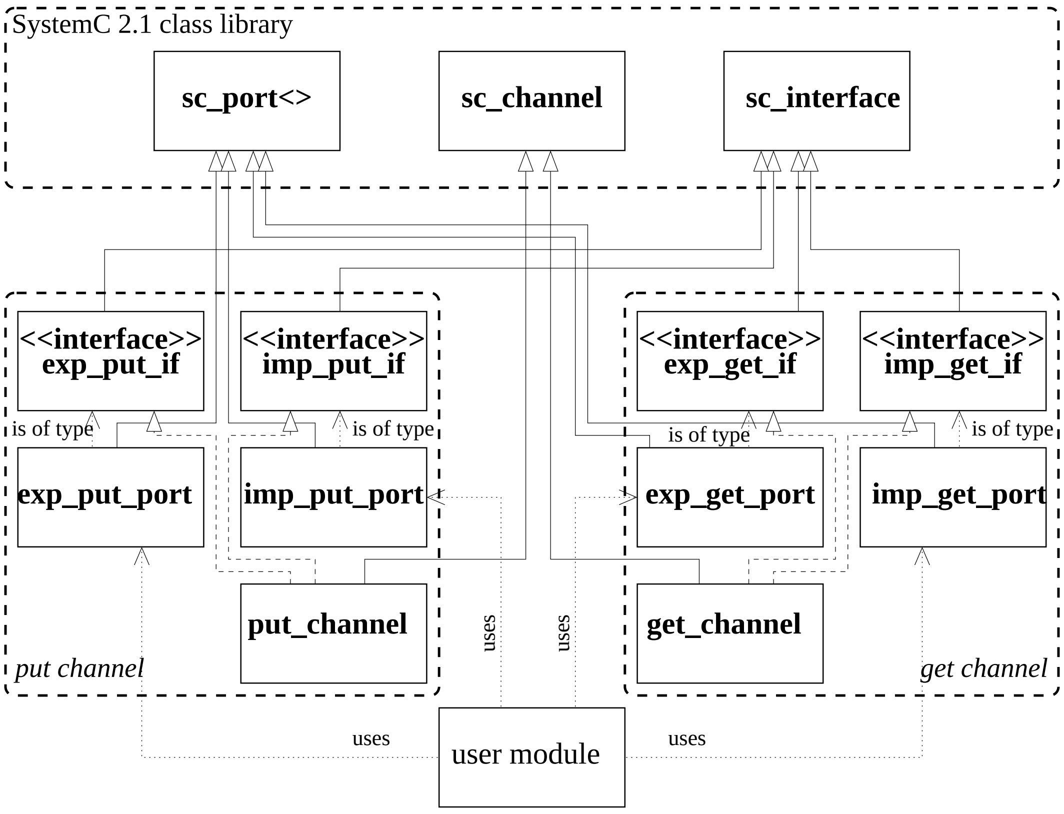

Figure 5: The class hierarchy of *put* and *get*.

only detect a state change in the communication channel, not the exact variable whose state has been changed. However, depending on which of the processes is executed first, it might be necessary to execute only one. That is, if the changed variable is found in the first executed process there is no need to execute the other. When a process is nondeterministically chosen for execution, that is, both processes sent a request at the same time, the guards of the process's component actions are evaluated and the winning component is executed.

5.2.3 Procedure Based Communication

As defined earlier, the interfaces of the procedure communication channels between a `Clerk` and `Bank` declare different sets of methods for these systems to call; `Bank` is able to control the information stored in the local variables of the channel, whereas `Clerk` is only allowed to request *Put* or *Get* procedures by using the corresponding methods provided by the channel interface. The class hierarchy of the ActionC *put* and *get* channels that implement the *Put* and *Get* procedures, respectively, is shown in Fig. 5.

Let us next elaborate the functionality of the Action Systems *Put* and *Get* procedures by viewing their implementation in ActionC, the *put* and *get* procedure commmunication channels. Import ports for these channels are constructed in the module body of `Clerk`, while the corresponding export ports are constructed in the module body of `Bank`. `Clerk` may commence a deposit or a withdrawal by calling the `put()` or `get()` method, respectively, which is provided by the interface between the `Clerk`'s port and the channel. The methods take as arguments the amount of money (`x`) and the bank account number (`addr`), that is, data and its destination address. The methods store this information on the channel's variables, and then inform `Bank` by sending an event notification for the SystemC scheduler. After being invoked, `Bank` reads the sent information from the channel by using the methods provided by the interface between `Bank` and the channel. Depending on the called method `Bank` either deposits into or withdraws from a bank account the amount of money, pointed out by the passed information. After this, `Bank` calls the channel's function to produce an event notification in order to inform `Clerk` that the deposit transaction (*Put* procedure) is complete, whereas the withdrawal transaction (*Get* procedure) is ended by obtaining the money using the `readX()` method provided by the `Clerk`'s interface.

5.2.4 Complex Interface Variables

The variable based communication channel, *comchannel*, is used in communication between the clerks and the `Arbiter` module. In contrast to *put* and *get* channels, *comchannel* has identical port-interface couples and both counterparts own the same access rights for the channel's methods. *comchannel* contains a pair of local boolean variables, `request` and `acknowledge` that in the Action Systems model correspond to the boolean variables *req* and *ack*, respectively. They are used in the signalling that follows the 4-phase handshaking protocol, in which the request and acknowledgement phases are initiated by first activating the request signal after which the acknowledgement signal is activated as a sign of opened access through channel. After the transaction has finished the handshaking protocol is finalised by initialising the request and acknowledgement signals again in that order. In this experiment, the clerks control the `request` variable to request a service, while `Arbiter` controls the `acknowledge` variable, thus guarding the access to the shared resource, the bank.

5.2.5 Source Files

All the source files created for the ActionC model of the presented bank office are gathered in Table 2. In addition to the `main` file the implementation includes the header(`.h`) and definition(`.cpp`) files for each of the four distinct action systems in the Action Systems model. Action Systems clerks A and B are implemented as two instances of a single `Clerk` module. There are also header and definition files for an ActionC communication channel `comchannel` that corresponds to the interface variable type *comchannel*, and for ActionC procedure communication channels *put* and *get* corresponding to the *Put* and *Get* procedures, respectively.

Table 2: Source files for the bank office example.

`main.cpp`	The main program file including the `sc_main()` routine.
`simconstants.h`	Simulation specific constants
`actionc.h`, `actionc.cpp`	ActionC methods
`entrance.h`, `entrance.cpp`	Implementation of $\mathcal{E}ntrance$
`clerk.h`, `clerk.cpp`	Implementation of $\mathcal{C}lerkX$
`arbiter.h`, `arbiter.cpp`	Implementation of $\mathcal{A}rbiter$
`bank.h`, `bank.cpp`	Implementation of $\mathcal{B}ank$
`put_procedure.h`, `put_procedure.cpp`	Implementation of *Put* procedure
`get_procedure.h`, `get_procedure.cpp`	Implementation of *Get* procedure
`comchannel.h`, `comchannel.cpp`	Implementation of *comchannel* type interface variable

The simulation specific constants are gathered in file `simconstants.h` and the declarations and definitions of the previously introduced ActionC methods are in files `actionc.h` and `actionc.cpp`, respectively.

5.3 Simulation

To run simulations on the ActionC bank office model the SystemC class library version 2.1 was installed on DLL release version 1.5.21-2 of Cygwin [13] environment running on Windows XP. The created model was then compiled with g++ compiler version 3.4.4-3 for Cygwin.

The Action Systems model describes a system that runs for an unlimited period of time, and the simulation of the created ActionC model can be implemented for similar behaviour. For practical reasons, however, we set the simulation to stop after a predefined number of transactions. In several parts of the simulation, the execution is monitored with printed screen outputs. Although printed reports are a more primitive approach to observe the model during simulation than a monitoring solution with SystemC trace files, it is also a more straightforward approach. This decision was made to keep the implementation as simple as possible, which is also the reason for allowing the presence of only one customer at a time inside the bank office. The model can be adapted for multiple customers by modifying the implementation of the $\mathcal{E}ntrance$ system. However, the simpler implementation used here is adequate enough to fulfil our goals to experiment nondeterminism and procedure based communication in ActionC.

The simulation execution is monitored in several parts of the model starting from the entrance of the first customer and lasting until the last customer has left the bank office. To keep the simulation output short, the execution is stopped after the third customer has done his business in the bank. The number of customers and the length of the simulation can be altered by changing the value of the constant `TRANSACTIONS` in file `simconstants.h`. With the setting:

```
const int TRANSACTIONS = 3;
```

the execution of the main program produced the following output:

```
Entrance(exec): Bank is now open
____Entrance(NewCustomer): Customer 3 enters the bank____
Entrance(QueueA): Customer 3 queues for ClerkA
Clerk(Customer): Clerk will perform a withdrawal
Clerk(ReqAccess): Clerk is requesting access to the bank accounts
Arbiter(ClerkA): ClerkA has been granted access to the bank accounts
Clerk(Withdrawal): Withdrawing sum 421 from account 3
Bank(Load): Withdrawing sum 421 from account 3
Bank(Load): The balance of account 3 is now 579
Clerk(Withdrawal): Received sum 421
Clerk(RelAccess): Clerk is releasing access to the bank accounts
Arbiter(ClerkA): Bank account access of ClerkA has been cancelled
____Clerk(WaitNext): Customer 3 exits the bank____
____Entrance(NewCustomer): Customer 2 enters the bank____
Entrance(QueueB): Customer 2 queues for ClerkB
Clerk(Customer): Clerk will perform a withdrawal
Clerk(ReqAccess): Clerk is requesting access to the bank accounts
Arbiter(ClerkB): ClerkB has been granted access to the bank accounts
Clerk(Withdrawal): Withdrawing sum 815 from account 2
Bank(Load): Withdrawing sum 815 from account 2
Bank(Load): The balance of account 2 is now 185
Clerk(Withdrawal): Received sum 815
Clerk(RelAccess): Clerk is releasing access to the bank accounts
Arbiter(ClerkB): Bank account access of ClerkB has been cancelled
____Clerk(WaitNext): Customer 2 exits the bank____
____Entrance(NewCustomer): Customer 5 enters the bank____
Entrance(QueueA): Customer 5 queues for ClerkA
Clerk(Customer): Clerk will perform a withdrawal
Clerk(ReqAccess): Clerk is requesting access to the bank accounts
Arbiter(ClerkA): ClerkA has been granted access to the bank accounts
Clerk(Withdrawal): Withdrawing sum 700 from account 5
Bank(Load): Withdrawing sum 700 from account 5
Bank(Load): The balance of account 5 is now 300
Clerk(Withdrawal): Received sum 700
Clerk(RelAccess): Clerk is releasing access to the bank accounts
Arbiter(ClerkA): Bank account access of ClerkA has been cancelled
____Clerk(WaitNext): Customer 5 exits the bank____
Entrance(exec): Bank is now closed
```

where on each line the first print before the parantheses indicates the module and the print inside the parentheses the executed action that produces the output line. Owing to the nondeterministic choices in several parts of the model, the course of the simulation and the output is, in all probability, always different. Similarly, as the number of customers, also the number of accounts in the bank is freely adjustable by altering the value of the constant `ACCOUNTS` in file `simconstants.h`.

This does not change the total length of the simulation but regulates the variety of possible customers entering the bank office during the simulation.

The operation of one loop of simulation begins from the `Entrance` module, where the next customer and the clerk that the customer will be queueing for are nondeterministically chosen by action process `newCustomer()` and the `exec()` process, respectively. Depending on the queue, either action process `queueA()` or `queueB()` then activates the `exec()` process of the corresponding `Clerk` module and sends the clerk the customer number through the integer type primitive channel between the modules. After being activated the `Clerk` module's `exec()` process executes action process `customer()`, which nondeterministically chooses between the deposit and withdrawal transactions. Next the clerk requests access to the bank accounts from the `Arbiter` module with action process `reqAccess()`. If both clerks are trying to request the access simultaneously, `Arbiter` makes the choice between them nondeterministically. In this simulation, however, there is only one customer in the bank office at a time, which also means that only one of the clerks is requesting the access during each simulation loop. Therefore, despite the `Arbiter` module's ability to manage simultaneous requests, none will appear due to the simplicity of this simulation. After being granted the bank access, the clerk performs the chosen transaction through the corresponding channel, that is, by using either *put* or *get* channel for deposit and withdrawal, respectively. The `Bank` module then executes either action process `save()` or action process `load()` depending on the case. Once the transaction has finished, the `Clerk` module's `exec()` process executes action process `relAccess()`, which releases the bank account access to be dealt out again by `Arbiter`. By activating action process `waitNext()` the `Clerk` module informs `Entrance` that it is ready to receive the next customer. After this, `Entrance` continues by dealing out the next customer for one of the `Clerk` modules if there are more transactions to be performed in the simulation.

6 Conclusions and Future Challenges

In ActionC the formal correct-by-construct development paradigm Action Systems and the industry standard design language SystemC integrate into an embedded computer system development framework. Action Systems includes many similar constructs as SystemC. Both Action Systems and SystemC use a modularised model structure and support modularisation mechanisms such as procedures, parallel composition and data encapsulation. Both can be used in describing entire HW/SW systems starting from an initial behavioural model and resulting with an implementable design that includes both hardware and software partitions of the system. For both languages, there are accurate rules concerning the refinement performed on the models. In addition, SystemC includes a simulation kernel that can be used in testing the created models.

In this report, the most important Action Systems aspects in the ActionC development have been nondeterminism and communication between system components. The introduced case study exemplified the use of the proposed ActionC framework and concentrated on these aspects. Nondeterministic choice in different parts of the model was implemented in several different ways depending on the proposed functionality in each case. The model that presented the activities of a bank office consisted of four distinct modules whose communication had to be organised by using primitive as well as hierarchical channels. The used hierarchical channel was the ActionC communication channel, and the specialisation of that channel, the ActionC procedure communication channel, was used to implement the procedure based communication in the initial Action Systems model. Being based on interfaces and procedure calls, the ActionC implementation of inter module communication resembles the accuracy aspects of a transaction level model. Therefore, based on the implementation of communication channels the best place for the bridge between Action Systems and SystemC is at the transaction level. The internal implementation of ActionC modules is quite straightforward because of the similarity in the structures of action systems and SystemC modules. For the internal implementation of modules, in addition to the first ActionC methods, an ActionC coding style has been introduced. By following the proposed coding style with the rules of transaction level modelling in the conversion process from Action Systems to SystemC, an initial ActionC model can be described, from which the design process can be continued by refining the model down to the desired levels of abstraction.

The ActionC framework underpins the Action Systems development framework by the simulation support provided by SystemC. The future development of ActionC should also include the objective to achieve direct logic synthesis from SystemC descriptions of Action Systems models. However, the possibility of synthesis based on SystemC is limited to the features included in its synthesisable subset. The future development of the ActionC framework should also ensure that the formal correctness of an Action Systems description is preserved in the simulations. This is especially useful when working on the first, high abstraction level model: An executable ActionC model of the initial specification provides valuable information for later refinement of the design. The decision to map Action Systems communication practises into SystemC at transaction level also requires further research. Timing is another aspect that the future development of the ActionC framework should also address.

The continuous development of the SystemC class library brings also challenges to the system designer that uses the environment. Features that have been valid in the previous release may well be deprecated in the next. Although, as the development advances, SystemC class library offers a more and more precise tool for modelling HW/SW systems, this comes with the cost of weaker backward compatibility. In the case of the bank office example, deprecated features, or should we say improved realism, brings up some implementational challenges, when SystemC class library is upgraded from version 2.1 to the next, version 2.2 beta. By default, the former version has no objections with multiple ports writing to one `sc_signal` instance, while the latter would deny such actions and suggest using other signal types in the place of `sc_signal`. Although, this deprecated feature can still be turned on if needed, these types of changes in the language cause problems for a system designer, and may be even more confusing in the development of a framework that uses SystemC as its implementation.

References

[1] K. Sere and R.-J. Back, “From Action Systems to Modular Systems,” in *FME’94: Industrial Benefit of Formal Methods*. Springer-Verlag, 1994, pp. 1–25.

[2] R.-J. Back, “A Calculus of Refinements for Program Derivations,” *Acta Informatica*, vol. 25, pp. 593–624, 1988.

[3] The Open SystemC Initiative, *SystemC Version 2.0 User’s Guide. Update for SystemC 2.0.1*, 2002.

[4] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*. Kluwer Academic Publishers, Boston / Dordrecht / London, 2002.

[5] B. Stroustrup, *The C++ Programming Language*, 3rd ed. Addison-Wesley Publishing Company, Reading, Massachusetts, USA, 1997.

[6] R.-J. Back and K. Sere, “Action Systems with Synchronous Communication,” in *PROCOMET*, 1994, pp. 107–126.

[7] R.-J. Back and R. Kurki-Suonio, “Decentralization of Process Nets with Centralized Control,” in *PODC ’83: Proceedings of the second annual ACM symposium on Principles of distributed computing*. ACM Press, 1983, pp. 131–142.

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

[9] R.-J. Back, *Correctness Preserving Program Refinements: Proof Theory and Applications*, ser. Mathematical Center Tracts. Mathematical Centre, 1980, vol. 131.

[10] J. Plosila, P. Liljeberg, and J. Isoaho, “Modelling and refinement of an on-chip communication architecture,” *Lecture Notes in Computer Science (LNCS)*, vol. 3785, pp. 219–234, 2005.

[11] The Open SystemC Initiative, “www.systemc.org (verified 2007-04-12).”

[12] ——, *Draft Standard SystemC 2.1 Language Reference Manual*, 2005.

[13] The Cygwin Development/Support, “www.cygwin.com (verified 2007-04-12).”

ISBN 978-952-12-2025-8
ISSN 1239-1891