Tomi Metsälä | Tomi Westerlund | Juha Plosila

Introducing Action Systems Class Hierarchy to SystemC Modelling

Turku Centre for Computer Science

TUCS Technical Report

No 946, June 2009

Introducing Action Systems Class Hierarchy to SystemC Modelling

Tomi Metsälä
Tomi Westerlund
Juha Plosila
University of Turku, Department of Information Technology
Joukahaisenkatu 3-5 B, 20520 Turku, Finland
topeme@utu.fi | tovewe@utu.fi | juplos@utu.fi

TUCS Technical Report

No 946, June 2009

Abstract

ActionC is an integration of SystemC, an informal design language for embedded systems, and Action Systems, a formal modelling language that supports verification and stepwise correctness-preserving refinement of system models. In the ActionC approach Action Systems sets a formal foundation for SystemC modelling providing a possibility to verify the design as early as during the construction of the first transaction level model. This report provides an implementation for the ActionC language structures that concern Action Systems type system verification with invariants. The implementation introduced here will be a part of a larger modelling framework that will be elaborated in the future.

Keywords: SystemC, Action Systems, ActionC, Formal methods, System modelling, System verification

TUCS Laboratory
Distributed Systems

1 Introduction

The concept of ActionC[1] integrates the formal correct-by-construct development paradigm of Action Systems[2] and the industry standard design language SystemC[3][4] into an embedded computer system development framework. Both Action Systems and SystemC use a modularised model structure supporting mechanisms such as procedures, parallel composition and data encapsulation. Both languages can be used in describing entire HW/SW systems starting from an initial behavioural model and resulting with an implementable design including both hardware and software partitions. For both languages, there are methods for refining the created models. SystemC also includes a simulation kernel that can be used in testing the models. SystemC is a modelling language that does not itself include any formal modelling or verification features. The purpose of ActionC is to fill this gap in SystemC. SystemC is a class library written in C++ and the same applies here to the ActionC implementation, which is written in C++ and is to be used alongside SystemC.

This report introduces an implementation that models the Action Systems type of constants, variables and invariants in SystemC. In Action Systems, invariants are clauses that dictate the legal states of the system at every point of time. In contrast to the static way of Action Systems, the modelled structures are here utilised in run-time checking of system variable values, which in any case, acts as a good introduction to the usage of the implemented structures. This work has been the first step in implementing several Action Systems features to be used with SystemC. The implementation introduced in this report has been tested with SystemC version 2.2.

The Action Systems formalism is introduced briefly in Sect. 2 with a short description of the language and an introduction to its verification mechanisms. Section 3 introduces SystemC by presenting its basic structures and their usage. Section 4 presents the concept of ActionC, which combines Action Systems and SystemC together. The matching language constructs between them are introduced briefly defining a foundation for the ActionC implementation. A C++ class hierarchy that implements Action Systems type of constants, variables and invariants in SystemC is presented in Sect. 5. These structures are then used in testing the state of a running system in Sect. 6. Section 7 provides concluding remarks.

2 Action Systems

Action Systems is a formal language for modelling, verifying and refining designs of hardware and embedded systems[2]. The Action Systems formalism was initially proposed by Ralph-Johan Back and Reino Kurki-Suonio[5] and it is based on *the guarded command language* by Edsger W. Dijkstra [6]. With Action Systems a system can be designed based on its logical behaviour, while the imple-

mentational decisions can be postponed until later stages of design. An action system is a program in which the system execution is described in terms of *atomic actions*. Once an atomic action is chosen for execution it is executed to completion without interference from other actions in the system. Only the initial and final states of an atomic action are observable meaning there are no observable states between them. If two actions do not share any variables, it is possible to execute them in parallel. In this case parallel and sequential executions of these actions are guaranteed to produce identical results.

An action system is a modular unit that has its own local variables and a single iteration statement that operates the execution of the atomic actions in the system. An action system may also include parameterised procedures and nested action systems. Action systems communicate either by exchanging information through shared variables or by using shared actions and remote procedure calls. This procedure based communication is accomplished by importing and exporting variables and procedures within the system module interface.

The correct behaviour of an action system is ensured with invariants and protocols, which express constraints on the system. Invariants define the legal states of the system as predicates on the local and global variables, while protocols define the set of legal actions in the system. That is, invariants dictate rules for system states and protocols dictate rules for the transitions between them.

2.1 Actions

Actions are defined (for example) by:

$A ::= \ abort$	(*abortion, non-termination*)
$\mid skip$	(*empty statement*)
$\mid x := e$	(*(multiple) assignment*)
$\mid \textbf{do}\ A\ \textbf{od}$	(*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*)
$\mid \lvert[\textbf{var}\ x := x_0;\ A]\rvert$	(*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:

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

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:

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

In the case of a guarded action $A \mathrel{\widehat{=}} p \rightarrow B$, 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.

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 M (imp p_I; exp p_E; )( g; )
|[
  private procedure
    p(in x; out y) : (P);
  public procedure
    p_E(in x; out y) : (P_E);
  constant
    c;
  variable
    l;
  action
    A_i : (aA_i);
  invariant
    I;
  protocol
    Pr;
  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 global variables or interface 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. Invariants I and protocols Pr express constraints on these 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** loop, actions are selected for execution based on their composition and enabledness. This is continued until there are no enabled actions, whereupon the computation suspends leaving the system in a state in which it waits for an external impact that would enable its actions again. Hence, an action system is essentially

an initialised block with a body that contains a repeatedly executing statement.

3 SystemC

SystemC[3][4] is a C++ based system modelling language that does not include any formal features but it has a verification library to be used in testing the created system models. SystemC can be used in writing a specification for a system that includes both hardware and software components. It is a class library for the standard C++ programming language [7], and it is an open source standard, which is currently supported and advanced by the Open SystemC Initiative (OSCI) [8].

SystemC can be used in creating cycle-accurate models of software algorithms, hardware architectures and interfaces of SoC and system-level designs. Even though it is built on a high-level software programming language, SystemC is also a suitable tool for hardware modelling. In addition to all the object-oriented features and development tools of standard C++ SystemC provides system architecture constructs not included in the standard, such as hardware timing, concurrency, and reactive behaviour. Both software and hardware partitions of a system model can be written in a single high-level language. Therefore, software and hardware partitions, both being written in SystemC, can also be tested using one common test bench and without the need for language conversions between different abstraction levels.

3.1 A SystemC Model

A SystemC model consists of SystemC *modules* that form locally and independently operating units. Breaking a design model into several small blocks 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 also contain a hierarchy of other modules. Modules run processes that describe their actual behaviour. Processes are triggered by events.

Modules are connected to each other by channels, which are used in inter-module and inter-process communication. There are two types of channels: *primitive channels* and *hierarchical channels*. A primitive channel 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. In practise, a hierarchical channel is a normal SystemC module that is redefined as a channel.

Modules access channels through ports and exports. 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 and use the channel. This way the port acts as an intermediary for the module, while the inter-

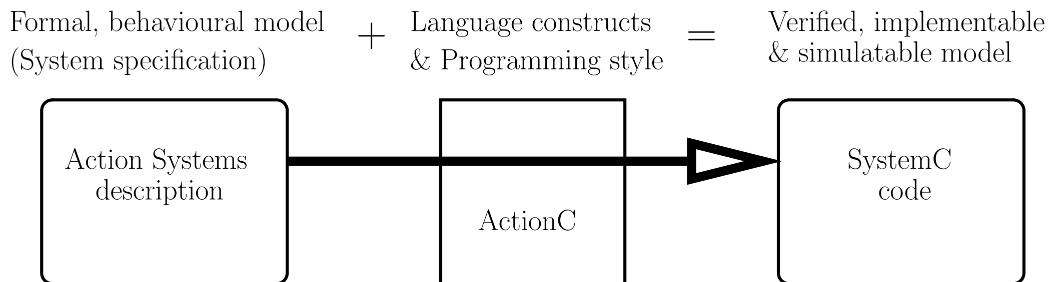


Figure 1: The concept of ActionC.[9]

face does the same for the channel. 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 during which the top level modules, channels and clocks are instantiated and module ports bound to the channel instances. Constructs inside the module hierarchies and hierarchical channels are instantiated in the module and channel constructors. Instantiation starts from the top and advances recursively through the hierarchies. 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 ActionC, Action Systems and SystemC are put together with the objective to utilise the best assets of both system modelling languages. Action Systems as a formal language is useful especially in the beginning of a system development process when the behaviour of a system should be defined as accurately as possible. SystemC is a powerful tool in simulation and implementation of a system model. 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, in some cases, synthesisable within the limits set by the synthesisable subset of SystemC (Fig. 1).

SystemC and Action Systems share several language constructs. The directly mappable constructs are gathered in Table 1. Both languages use modular constructs in creating local scopes: SystemC models consist of modules, while an action system is the corresponding structure in the Action Systems formalism. In ActionC it is called an *ActionC module*. The atomic actions of Action Systems are implemented as member functions of the ActionC module being either normal C++ methods or SystemC processes. These member functions are called *ActionC actions* and they are executed by performing *action calls*. The **execution**

Table 1: Matching language constructs between Action Systems and SystemC.

Action Systems		**SystemC**
action system	$\Leftrightarrow$	Module
in variable	$\Leftrightarrow$	sc_in$\langle\rangle$
out variable	$\Leftrightarrow$	sc_out$\langle\rangle$
inout variable	$\Leftrightarrow$	sc_inout$\langle\rangle$
(local) **variable**	$\Leftrightarrow$	private C++/ SystemC variable
private procedure	$\Leftrightarrow$	private C/ C++ void method
public procedure	$\Leftrightarrow$	SystemC hierarchical channel
function	$\Leftrightarrow$	private C/C++ non-void method
action	$\Leftrightarrow$	Module member
execution loop	$\Leftrightarrow$	SystemC thread process
non-atomic sequence **;**	$\Leftrightarrow$	sequential execution

loop of each action system is also implemented as a thread process that runs the local actions one at a time. The thread is suspended while local actions and actions in other modules are executed. 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. In case of a more complex channel structure, a SystemC hierarchical channel is a more practical solution. An example of such case is the procedure based communication model of Action Systems, which is given a SystemC implementation, for example, in [9]. The nondeterministic choice $'\ [] \ '$ is an essential operator in an Action Systems model. An implementation for this structure in SystemC environment is introduced in [1].

5 Action Systems Class Hierarchy for SystemC

Action Systems structures can be introduced to SystemC by composing them into a C++ class hierarchy that can be used alongside SystemC classes. This hierarchy will consist of classes that model action systems and the declaration clauses of their features, for example, constants, variables, actions and procedures. Also

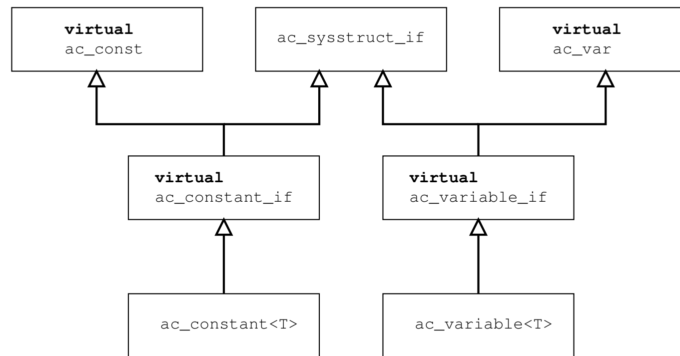


Figure 2: ActionC classes describing constants and variables.

the Action Systems type of data structures need to be implemented in addition to the verification constructs, that is, invariants and protocols. In this report the class hierarchy is referred to as ActionC class hierarchy and the classes as ActionC classes. The implementation introduced here has been tested with SystemC library version 2.2.

An action system is modelled with `ac_system` class. For every module in a SystemC model there is an instance of `ac_system` class. This instance is associated with the members of the action system corresponding to the SystemC module. `ac_system` objects may include other `ac_system` objects, which corresponds to nesting action systems. Every module member in a SystemC model is encapsulated inside a corresponding object in the ActionC class hierarchy. `ac_system` objects hold lists of the member structures that are associated with them. Each list corresponds to an Action Systems declaration clause. Lists contain pointers, for instance, to constants, variables, invariants, protocols and subsystems. These pointers are created and added to the lists during the initialisation of SystemC modules.

5.1 Constants and Variables

For a constant integer value, for example, an ActionC class object that describes a constant structure with integer as the type of the returned value is constructed as follows:

Action Systems: **constant** *max* : *Integer* := 13 ;
SystemC: `const int max = 13;`
ActionC: `ac_constant<ac_basictype<int> >* constants::max`
 = new ac_constant<ac_basictype<int> >("max","int",13,false);

In the ActionC approach `ac_constant<T>` is a template class that takes as its value an Action Systems type of data structure. The data structure can be either

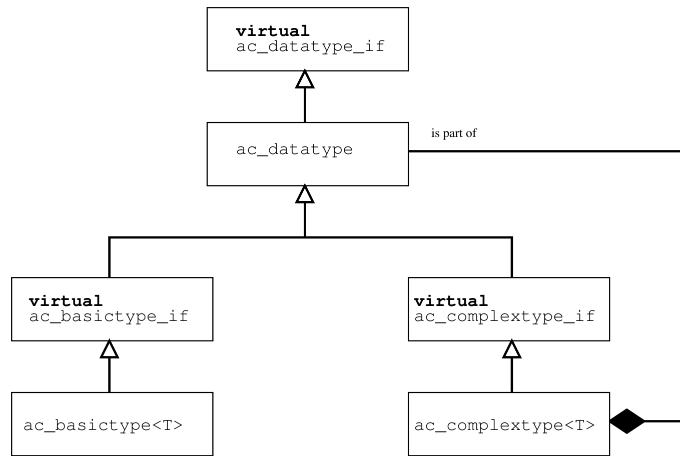


Figure 3: Classes describing Action Systems datatype.

a class inheriting from ac_complextype<T>, or a basic type ac_basictype<T> that holds a normal C++ compatible data type (Fig. 2). The same implementation model can be applied to Action Systems type of variables with the exception that the value of a variable can be modified after initialisation. In addition to a return value the objects of these classes are initiated with a name label and a string indicating the type of the return value. With the last attribute of the constructor a Boolean member variable is set or reset determining if the constant/variable is global or local.

Basic and complex types form a composite type object hierarchy where basic types are at the bottom of the hierarchy encapsulating the actual return value that the constant/variable holds (Fig. 3). Template parameter type T of an ac_complextype<T> object indicates the type of the next object in the hierarchy. The type is either ac_basictype<T> or a class that inherits ac_complextype<T>. Parameter type T of an ac_basictype<T> object is a C++ compatible data type, which is not an object, for instance, int, short or double. When calling (or changing) the value it is retrieved recursively through the object hierarchy from the ac_basictype<T> object at the bottom. Also, the type of the constant/variable is the type T of the ac_basictype<T> object, not the template parameter type of the constant/variable object itself.

Constants and variables, which are used by systems, are declared and defined separately in a container that they can be used from. Access to the constants and variables is given only for classes that are declared as friends of the container class. The container class does not include any public members, but all references to it are made by its friend classes. Constants and variables, as well as invariants, are linked to an ac_system instance with a pointer when the system is constructed (Fig. 4). The header file of a container class for variables used by system sys may look like this:

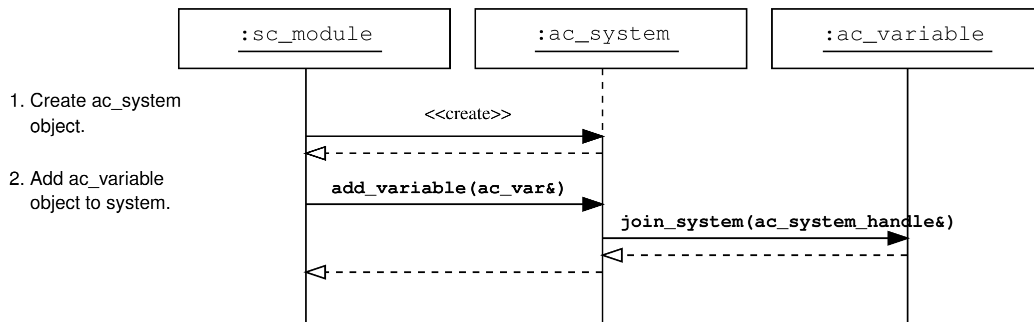


Figure 4: Creating an `ac_system` object and adding a variable into it.

```
#ifndef __sys_var_hpp__
#define __sys_var_hpp__

#include <ac_variable.h>
#include <ac_queue.h>

/** Container class for variables of system 'sys'. */
class sys_var {
  friend class Sys;             ///< The host SystemC module.
  friend class sys_invariant;   ///< User-defined invariant of system sys.
  friend class sys_clause1;     ///< User-defined clause.
  friend class sys_clause2;     ///< User-defined clause.

protected:
    static ac_variable<ac_basictype<short> >* varA;   ///< Declaring variable varA.
    static ac_variable<ac_queue<int> >* varB;         ///< Declaring variable varB.
};
#endif
```

The source file with the definitions may look as follows:

```
#include "sys_var.h"

  /** Values. */
    /** Constructor parameters of a basic type: name label, value. */
  ac_basictype<short>* valA = new ac_basictype<short>( "valA", 0 );
    /** Constructor parameters of a complex type: name label. */
  ac_queue<int>* valB = new ac_queue<int>("valB"); ///< Inherits from ac_complextype<T> class.

  /** Variables. Constructor parameters: name label, return type, value, global/not global. */
  ac_variable<ac_basictype<short> >* sys_var::varA
    = new ac_variable<ac_basictype<short> >("varA","short", valA, false);
  ac_variable<ac_queue<int> >* sys_var::varB
    = new ac_variable<ac_queue<int> >("varB","int", valB, false);
```

`ac_queue<T>` is a template class that implements the Action Systems datatype *queue*, which is a FIFO type structure. In this case the queue holds data that is of type integer or compatible.

5.2 Invariants and Clauses

Invariants in Action Systems are predicates on local and global variables of a system. They define constraints on the states of the system they are associated with.

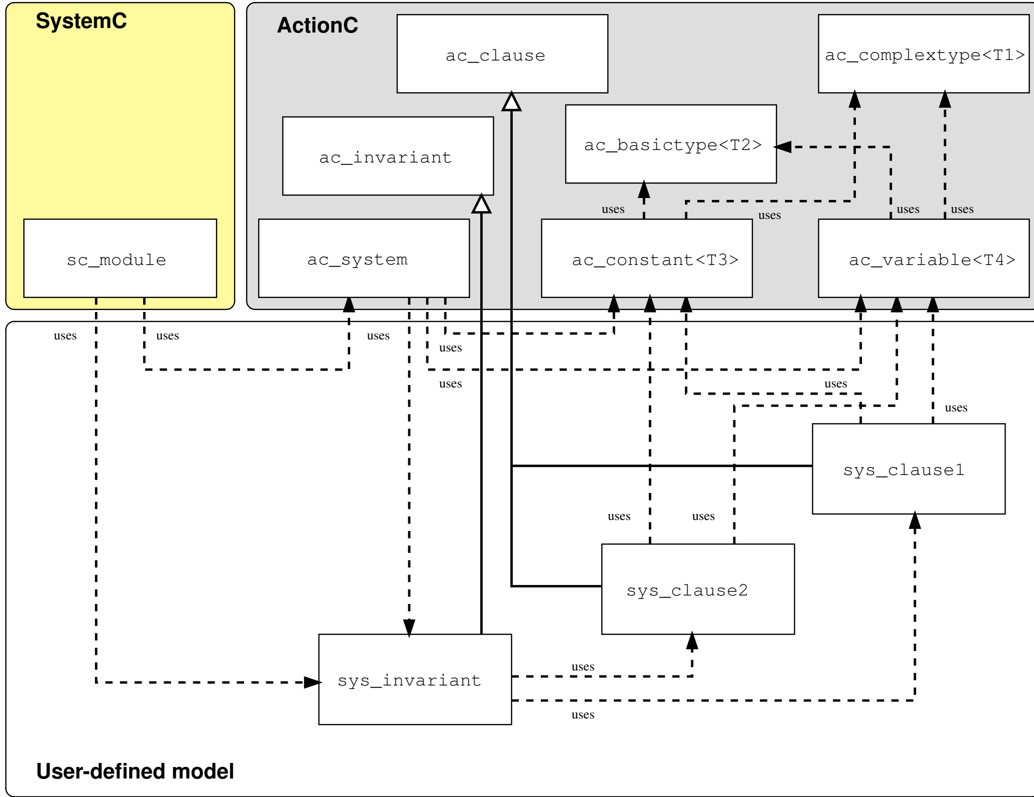


Figure 5: Class diagram of a system with an invariant of two clauses.

Constraints may be simple statements that set limits to the values of variables and constants within a specific system. Constraints can also be more complex Boolean expressions, relations between variables and constants. Predicate I is an invariant over an action A if it satisfies the condition

$$I \Rightarrow \mathbf{wp}(A, I)$$

Therefore, the action A preserves the invariant I. If

$$\mathbf{wp}(A, I) \equiv T$$

the action A is said to establish the invariant I. In practise, based on the logical implication above, action A preserves invariant I in all other cases except when I evaluates to true before A is executed and to false after the execution. Therefore, the preservation of an invariant depends on the system states both before and after action executions.

Actions in an action system preserve the *total invariant* I_{Tot} if they preserve all (n) invariants associated with that system:

$$I_{Tot} = I_1 \land I_2 \land \ldots \land I_n$$

This means that in order for the actions in an action system to preserve the total invariant, they must establish all the invariants at initialisation, and preserve them through every execution.

The invariants of Action Systems can be implemented in SystemC using the implementations of constants and variables. Invariants consist of *clauses* that are Boolean expressions including values of constants, variables and C++ data types. In the ActionC approach invariant is an instance of a class that can be used in testing the clauses associated with it. Clauses are gathered together and evaluated as one combined expression. An invariant evaluates to true when the combination of its clauses evaluates to true.

System's total invariant consists of all its individual invariants. The individual invariants consist of one or more clauses, which are boolean expressions with one comparison operator and two operands the operator compares. The operators can be values of C++ and SystemC data types. User-defined data types can also be used if the necessary operators are defined for them.

Invariants and clauses are added to a SystemC model as new C++ classes that are compiled along with the rest of the design that user creates. These user-defined invariant and clause classes inherit from ActionC classes `ac_invariant` and `ac_clause`, respectively (Fig. 5). Each `ac_system` instance that describes an action system for a corresponding SystemC module is used in creating the instances of the needed invariants.

For example, in Action Systems an invariant may dictate the following constraint for a system:

$$sys_invariant : (varA < constA) \wedge (head(varB) \leq constA) ;$$

where *varA* and *varB* are variables and *constA* is a constant used by the system. In SystemC a user-defined clause class `sys_clause1` that defines the first boolean expression, $varA < constA$, may look as follows:

```
/** Clause: left < right. */
template<class L, class R> class sys_clause1 : public ac_clause
{
  public:
    /** Constructor. Pointers to the operands are given as parameters. */
    sys_clause1( ac_sysstruct_if* l_p, ac_sysstruct_if* r_p ) { lo_p = l_p; ro_p = r_p; }

    /** Clause is executed.
     *  Clause retrieves the values of the handled elements and compares them.
     */
    virtual bool exec() {
      left = (dynamic_cast<ac_basictype<L>* >(lo_p->vtype()))->v();
      right = (dynamic_cast<ac_basictype<R>* >(ro_p->vtype()))->v();
      if ( left < right ) { return true; } else { return false; }
    }
  protected:
    L left;                 ///< Left operand of the clause.
    R right;                ///< Right operand of the clause.
    ac_sysstruct_if* lo_p;  ///< Pointer to the left operand.
    ac_sysstruct_if* ro_p;  ///< Pointer to the right operand.
};
```

The constructor of the invariant that uses `sys_clause1` and clause class `sys_clause2`, which defines expression $head(varB) \leq constA$ in SystemC, may look like this:

```
sys_invariant::sys_invariant( char* lab ) {
  label = lab;   ///< Set name label as lab.
  invh = ac_invariant_handle( label, this );

  /** Adding clauses to sys_invariant. */
  cl1 = new sys_clause1<short,int>(sys_var::varA, sys_const::constA); add_clause(cl1);
  cl2 = new sys_clause2<int,int>(sys_var::varB, sys_const::constA); add_clause(cl2);
}
```

The clauses that invariants test are instantiated by the invariant itself. The constructor is provided with the types and values of the variables and constants the clause uses. A clause checks the values of the constants and variables each time it is executed, that is, when the invariant that uses it, is run.

6 Checking Invariants in a Running Simulation

Let us now combine the small examples in Sect. 5 into one. Let us have action system $\mathcal{S}ys$ that has the form:

sys $\mathcal{S}ys$ (**in** $go : Boolean; in : Integer;$ **out** $out : Integer;$)
$|[$
  constant
    $constA : Integer := constA_{val};$
  variable
    $varA : Short; varB : queue\,of\,Integer;$
  action
    $Pop\colon (go \rightarrow varA = in; varB = varB \,@\, varA; out = varA);$
  invariant
    $sys_invariant : varA < constA \wedge head(varB) \leq constA;$
  initialisation
    $varA, varB := varA_{init}, varB_{init};$
  execution
    do Pop **od**
$]|$

This system includes one action, *Pop*, which is executed every time the value of the boolean type global variable *go* is true. Action *Pop* reads the value of global variable *in* and stores it on local variables *varA* and *varB*. At the end, the value of *varA* is copied onto global variable *out*. After this, *Pop* and the entire system $\mathcal{S}ys$ is suspended until the value of variable *go* is true again.

The header file of this system's implementation in SystemC may look like this:

```
#ifndef __sys_hpp__
#define __sys_hpp__

#include "sys_constants.h"            ///< File containing constants for Sys.
#include "sys_variables.h"            ///< File containing variables for Sys.
#include "sys_invariant.h"            ///< File declaring an invariant for Sys.
#include "ac_sysstruct/ac_system.h"   ///< Header file of ac_system class.

SC_MODULE(Sys) {

  /** System */
  ac_system* sys;            ///< The ac_system instance.
  sys_invariant* sys_inv;    ///< An instance of sys_invariant.

  /** Ports. */
  sc_in<bool> go;            ///< Port for activation signal from outside the system.
  sc_in<int> in;             ///< Port for signal bringing in integers.
  sc_out<int> out;           ///< Port for signal sending out integers.

  void pop();                ///< Action Pop.
  void execution();          ///< Execution loop.
  sc_signal<bool> pop_sig;   ///< Signal to control action Pop.

  SC_CTOR(Sys) {

    sys = new ac_system("sys");               ///< The instance of ac_system.

    /** Constants. */
    sys->add_constant(*constants::constA);    ///< Add constA to sys.

    /** Variables. */
    sys->add_variable(*variables::varA);      ///< Add varA to sys.
    sys->add_variable(*variables::varB);      ///< Add varB to sys.

    /** Invariants. */
    sys_inv = new sys_invariant("sys_inv");   ///< The instance of sys_invariant.
    sys->add_invariant(*sys_inv);             ///< Add sys_inv to sys.

    /** Process declarations. */
    SC_THREAD(execution);
    sensitive << go; set_stack_size(0x50000); dont_initialize();

    SC_THREAD(pop);
    sensitive << pop_sig.posedge_event(); set_stack_size(0x50000); dont_initialize();

    sys_inv->test();                          ///< Test invariant at initialisation.
  }
};
#endif
```

The source file with the method definitions may look as follows:

```
#include "sys.h"

/** Execution loop handling action Pop. */
void Sys::execution() {

  while(true) {
    sys_inv->test();                                ///< Test invariant before action Pop.
    pop_sig.write(true); wait(pop_sig.negedge_event());   ///< Execute action Pop.
    sys_inv->test();                                ///< Test invariant after action Pop.

    wait(go.posedge_event());                       ///< Wait for next activation from port 'go'.
  }
}
```

```
/** Action Pop. */
void Sys::pop() {

  while(true) {
    /** Receiving integer from port 'in'. */
    (dynamic_cast<ac_basictype<short>* >(variables::varA->vtype()))->set_value(in->read());
    /** Saving integer into varB. */
    (dynamic_cast<ac_queue<int>* >(variables::varB->vtype()))
      ->append((dynamic_cast<ac_basictype<short>* >(variables::varA->vtype()))->v());
    /** Passing integer forward through port 'out'. */
    out.write((dynamic_cast<ac_basictype<short>* >(variables::varA->vtype()))->v());

    /** Wait for next activation from signal 'pop_sig'. */
    pop_sig.write(false);
    wait();
  }
}
```

As can be seen from the code, invariant `sys_inv()` is tested by running method `test()` during initialisation as well as before and after running `pop()`.

In the Action Systems formalism an action system preserves its total invariant on the basis of the weakest precondition semantics. In the current ActionC implementation, however, invariants are tested during simulation. In order to verify the correctness of a running system its invariants should be tested at initialisation as well as before and after every execution of its actions. The result of an invariant test is revealed by the results of each individual clause execution. If all clauses evaluate to true, also the invariant test evaluates to true at that point of simulation. If at least one clause evaluates to false, also the invariant test evaluates to false indicating that the system does not run properly. However, in Action Systems an action preserves an invariant in all other cases except when it evaluates to true before the action is executed and to false after the execution. Therefore, the preservation of an invariant depends on tests both before and after action executions. This kind of runtime invariant checking does not verify that a system is correct by construction, but it can be used as a tool in the modelling process. Introducing a tool for creating systems that are correct by construction will be a challenge for future ActionC implementations.

Designer can write commands into an invariant about further measures that will be taken after passing or failing tests. `ac_invariant` class includes methods `test_passed()` and `test_failed()`, which can be overridden in the user-defined invariant class (Fig. 6). The definitions of these methods in `ac_invariant` class do nothing so they must be overridden if any measures need to be performed by them in a user-defined model.

7 Conclusion

Both the Action Systems formalism and the SystemC modelling environment use a modularised model structure supporting mechanisms such as procedures, par-

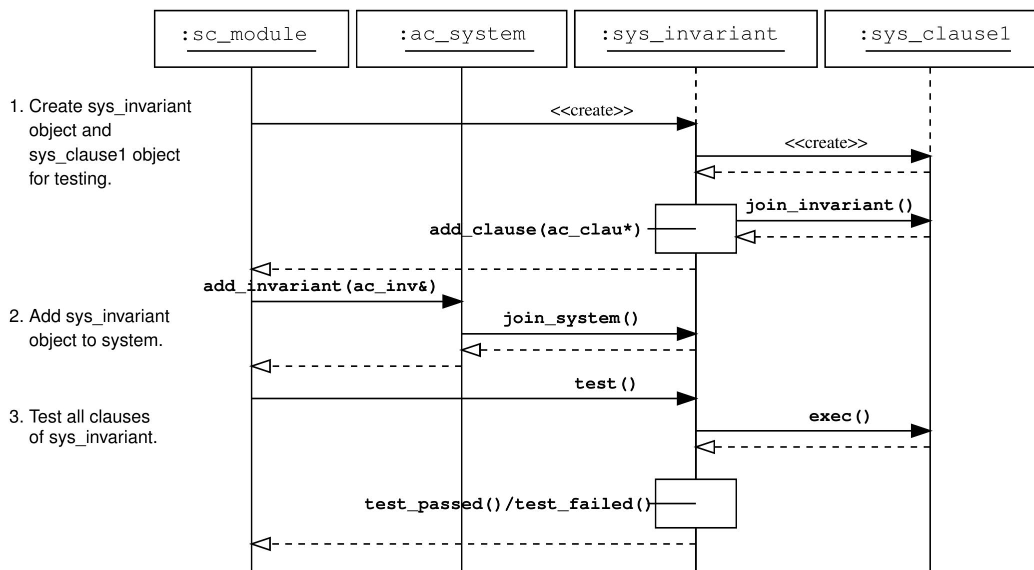


Figure 6: Creating an object of user-defined `sys_invariant` class, adding it into a system and executing the `sys_clause1` object in it.

allel composition and data encapsulation. Both can be used in describing behavioural models of HW/SW systems and refining them in a stepwise manner down to an implementable design. In ActionC these two languages integrate into an embedded computer system development framework, which enables mathematically rigorous, correct-by-construct modelling with simulation support for running the created models.

This report has introduced an implementation for an Action Systems class hierarchy that is to be used in SystemC modelling. In Action Systems, invariants express constraints on the system they are associated with. They ensure that the system behaves correctly by defining its legal states as predicates on its local and global variables. In this report, however, the use of the implemented structures have been demonstrated by utilising them in checking the values of system variables at fixed points of simulation. This kind of run-time invariant checking can be used as a tool in system modelling process but it does not directly adhere to the static verification approach of Action Systems. That approach will be tackled in the future implementations of ActionC.

References

[1] T. Metsälä, T. Westerlund, S. Virtanen, and J. Plosila, "ActionC: An Action Systems Approach to System Design with SystemC," Turku Centre for Computer Science (TUCS), Turku, Finland, Tech. Rep. 865, Jan. 2008.

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

[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] 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.

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

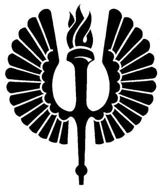
[7] B. Stroustrup, *The C++ Programming Language*, 3rd ed. Addison-Wesley Publishing Company, Reading, Massachusetts, USA, 1997.

[8] The Open SystemC Initiative, "www.systemc.org (verified 2009-01-29)."

[9] T. Metsälä, T. Westerlund, S. Virtanen, and J. Plosila, "Rigorous Communication Modelling at Transaction Level with SystemC," in *ICSOFT 2008: Proceedings of the Third International Conference on Software and Data Technologies*. INSTICC Press, 2008, pp. 246–251.

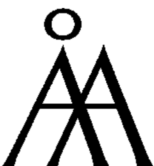
TURKU CENTRE *for* COMPUTER SCIENCE

Joukahaisenkatu 3-5 B, FI-20520 Turku, Finland | www.tucs.fi



University of Turku

- Department of Information Technology
- Department of Mathematics



Åbo Akademi University

- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration

- Institute of Information Systems Sciences

ISBN 978-952-12-2310-5
ISSN 1239-1891