RIGOROUS COMMUNICATION MODELLING AT TRANSACTION LEVEL WITH SYSTEMC

Tomi Metsälä[1,2], Tomi Westerlund[2], Seppo Virtanen[2] and Juha Plosila[2,3]
[1]*Distributed Systems Design Laboratory, Turku Centre for Computer Science, Turku, Finland*
[2]*Department of Information Technology, University of Turku, Turku, Finland*
[3]*Academy of Finland, Research Council for Natural Sciences and Engineering, Finland*

Keywords: Embedded Systems, Formal Methods, Action Systems, SystemC.

Abstract: We introduce a communication model for ActionC, a framework for rigorous development of embedded computer systems. The concept of ActionC is the integration of SystemC, an informal design language, and Action Systems, a formal modelling language supporting 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 that also includes a simulation environment. Translation of an Action Systems model to the corresponding ActionC model is carried out with the means provided by SystemC and in a way that preserves the semantics of the underlying formal model. Hence, the ActionC framework allows us to reliably simulate Action Systems descriptions using standard SystemC tools, which is especially important for validating the initial formal specification of a system. Our initial experiments with ActionC have successfully produced correct-proven simulatable SystemC descriptions of Action Systems.

1 INTRODUCTION

Verifying the correctness of a modern hardware or HW/SW system design claims more time and effort than ever before. Verification process of complex multiprocessor and reconfigurable systems brings up new challenges for both researchers and developers in the industry. Verification by testbench simulation is the prevailing approach to this issue. For this purpose, there exists a wide variety of recently introduced new verification languages, methods and tools. Another approach to a system verification task includes the use of formal methods. By specifying system in a formal design language developer is able to verify the model in a mathematically rigorous manner. Based on a formal specification the design is correct by construction and its logical behaviour does not need to be verified by simulation. However, the industry lacks the type of comprehensive tools that can be used in formal, stepwise development of embedded HW/SW systems throughout the design project all the way from abstract specification down to implementable model.

In order to address this challenge we introduce the idea of *ActionC* framework, which is the integration of the formal system specification language *Action Systems* (Sere and Back, 1994) and the informal modelling and simulation environment *SystemC* (The OSCI, 2005). Action Systems provides a rigorous method for specifying and verifying modular HW/SW systems in addition to the refinement calculus framework (Back, 1988), a stepwise development environment for refining an abstract specification down to a model that can be translated into a suitable implementation. On the other hand, SystemC is a programming library that is built on top of the standard C++ programming language (Stroustrup, 1997). In addition to the object oriented techniques provided by C++, SystemC offers the system designer hardware-oriented functionality such as clocks, signals, reactivity and modelling of parallel processing. With SystemC it is possible to gradually develop and verify a HW/SW system design in its entirety without the need to translate the hardware model into a separate hardware design language. In ActionC, Action Systems provides a formal foundation for SystemC modelling enabling the developer to produce a simulatable system description from an Action Systems description while preserving the formal correctness throughout the model's refinement process as well as during the simulation.

Inter-module communication is an essential feature in Action Systems modelling, and therefore it will be addressed in this paper. We elaborate the way in which Action Systems' procedure based communi-

Metsälä T., Westerlund T., Virtanen S. and Plosila J. (2008).
RIGOROUS COMMUNICATION MODELLING AT TRANSACTION LEVEL WITH SYSTEMC.
In *Proceedings of the Third International Conference on Software and Data Technologies - SE/GSDCA/MUSE*, pages 246-251
DOI: 10.5220/0001900402460251

cation model is implemented in the ActionC framework with the means provided by SystemC.

Several other papers have also covered the issue of improving SystemC by incorporating it with formal design and verification methods. A general review concerning SystemC and different formal verification method has been conducted for example by Vardi in (Vardi, 2007). The integration of an independent formal language into SystemC has been explored for example by Habibi and Tahar in (Habibi and Tahar, 2005) and by Patel et al. in (Patel et al., 2006). In the former a language called AsmL has been used for the verification of transaction level models of SystemC, while the latter integrates Bluespec and SystemC in order to create rule-based MoCs for SystemC. On the other hand, a tool for executing action systems has been tested by Degerlund, Walden and Sere in (Degerlund et al., 2007), in which they translated Action Systems models into C++ and executed the translation with an experimental scheduler tool.

The paper is organised as follows: Action Systems and SystemC are introduced in Sect. 2 and Sect. 3, respectively, and in Sect. 4 they are viewed together as the ActionC framework. Section 4 will also address the implementation of the procedure based communication. Concluding remarks are provided in Sect. 5.

2 ACTION SYSTEMS

Action Systems is a design framework that can be used in modelling complex parallel and reactive systems in a formal, mathematically rigorous manner (Sere and Back, 1994). The Action Systems formalism was initially proposed by Back and Kurki-Suonio (Back and Kurki-Suonio, 1983) and it is based on the *guarded command language* by Dijkstra (Dijkstra, 1976). In Action Systems a system model is described according to the system's logical behaviour. The behaviour is modelled in terms of *atomic actions*, which are guarded commands that perform an operation for a given set of system variables. Once chosen for execution, atomic actions are executed to completion without interference from other actions in the system. Only the initial and final states of an atomic action are observable in the environment. Parallelism between actions can be modelled when they do not share any variables that either or both of them write onto. In such cases actions can be executed in any order without affecting the final state.

The verification of total correctness of actions and entire systems is based on the concept of *weakest precondition*: for every action there exists a postcondition that can be established by properly executing and

```
sys F ifo ( exp Put(in x: Integer);
            exp Get(out x: Integer); )
          [ N: Natural := 8; ] ::
|[ variable
   q: queue of Integer ; cnt: 0...N;
  public procedure
   Put(in x: Integer): (q := q@⟨x⟩);
   Get(out x: Integer): (x,q := head(q),tail(q));
  action
   waitPut: (cnt < N → await Put; cnt := cnt + 1);
   waitGet: (cnt > 0 → await Get; cnt := cnt − 1);
  invariant
   Range: 0 ≤ cnt ≤ N;
  initialisation
   q,cnt := ⟨ ⟩,0;
  execution
   do WaitPut [] WaitGet od    ]|
```

Figure 1: Description of the action system $\mathcal{F}\,ifo$.

terminating the action statement. In that case the execution of that action is guaranteed to lead the system into a state that satisfies the weakest precondition. An initial system specification can be refined into an implementable model using a method called *refinement calculus* (Back and von Wright, 1998). The final model is developed in a stepwise manner preserving the verified correctness of the initial model. Therefore, the design can be seen as correct-by-construction at all stages of the process.

2.1 Action Systems Description

An *action system* is a module structure that offers an encapsulated environment for local variables and actions that operate on them. Actions also use interface procedures and variables in communicating with actions in other action systems. Private procedures are used only within the local action system scope. Execution of an action system is modelled with an **execution** loop that runs the actions in a specific order, which may or may not be deterministic.

As an example of an action system description, Fig. 1 illustrates a simple transaction level FIFO structure, in which we can identify the three main sections of the action system description: *interface*, *declaration* and *iteration*. In this case, the interface part declares *communication procedures Put* and *Get* that the action system $\mathcal{F}\,ifo$ introduces and exports. The use of communication procedures corresponds to a transaction level model with remote function calls. The interface also introduces parameter N, a natural number, that is set to value eight. The exported procedures are imported by other action systems that require the service that these procedures provide. Procedures are in general any atomic actions, possibly with some auxiliary local variables that are initialised every time the

procedure is called. The execution of a procedure is considered as a part of the calling action, and thus procedures can be considered as parametrisable subactions. If an action system does not have any communication variables or procedures, it is *a closed action system*, otherwise it is *an open action system*.

In the declaration part the local variables, actions as well as private and public procedures are defined. The $\mathcal{F}ifo$ action system defines the *Put* and *Get* procedures. *Put* appends (the @-operation) the element, given as a parameter, at the end of the queue type local variable *q*. *Get*, on the other hand, returns the procedure caller the head (the first element) of the queue, while setting the tail (all elements except the first) as the new value of the queue. Both actions *waitPut* and *waitGet* perform an **await** command that passively waits for a procedure call from an action system that imports the procedure. When a call comes, a procedure is executed with the rest of the statements in the action. The action *waitPut* increments the local variable *cnt* after executing *Put*, while the action *waitGet* executes *Get* and decrements the variable *cnt*. Both actions have their own *guard*, $gd()$, that must hold before the action's *body*, $bd()$, is allowed to execute, e.g. $gd(Put) \mathrel{\widehat{=}} cnt < N$ and $bd(Put) \mathrel{\widehat{=}}$ **await** *Put*; $cnt := cnt + 1$). The declaration part also defines the constraints that the actions must uphold. Here, a range for the value of the variable *cnt* is introduced as a system invariant. The operation of the action system is started by initialisation in which the variables are set to predefined values.

In the iteration part, that is, 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. The operation of the $\mathcal{F}ifo$ action system depends exclusively on the procedure calls coming from other systems because of the **await** commands in both of its two actions.

2.2 Procedure based Communication

As we learned in Sect. 2.1 the procedure based communication model uses interface procedures to model communication channels between action systems. Let us consider an action system $\mathcal{C}lient$ (Fig. 2) and the action system $\mathcal{F}ifo$ that was introduced earlier. Action system $\mathcal{C}lient$ now imports the communication procedure *Get*, which $\mathcal{F}ifo$ exports. The action *pop* of the action system $\mathcal{C}lient$ uses the imported *Get* procedure to retrieve the first element from the FIFO (Fig. 3). The communicating actions *pop* of $\mathcal{C}lient$

```
sys Client (in on: Boolean; out result: Integer;
            imp Get(in x: Integer); )
|[ variable
    a: Integer;
  action
    pop: (on → call Get(a); result := a);
  initialisation
    result := 0;
  execution
    do pop od    ]|
```

Figure 2: Description of the action system $\mathcal{C}lient$.

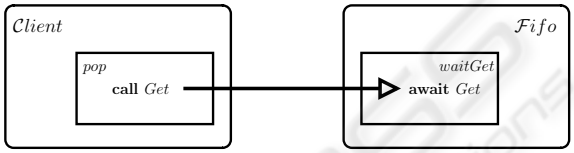

Figure 3: Action system $\mathcal{C}lient$ retrieving data from action system $\mathcal{F}ifo$ using the communication procedure *Get*.

and *waitGet* of $\mathcal{F}ifo$ are:

$$pop \mathrel{\widehat{=}} (pop_1; \textbf{call } Get(a); pop_2)$$
$$waitGet \mathrel{\widehat{=}} (waitGet_1; \textbf{await } Get; waitGet_2)$$

where $pop_{1,2}$ and $waitGet_{1,2}$ are subactions of *pop* and *waitGet*, respectively. In this example, $pop_2 \mathrel{\widehat{=}} result := a$ and $waitGet_2 \mathrel{\widehat{=}} cnt := cnt - 1$, while $waitGet_1 \mathrel{\widehat{=}} cnt > 0 \rightarrow skip$ and $pop_1 \mathrel{\widehat{=}} on \rightarrow skip$, where *skip* is an empty action. Furthermore, the local variables of the systems are distinct, $l_{\mathcal{C}lient} \cap l_{\mathcal{F}ifo} = \emptyset$, communication variables are a set $g_{\mathcal{C}lient} \cap g_{\mathcal{F}ifo}$ and the initialisations of the communication variables $g_{\mathcal{C}lient} \cap g_{\mathcal{F}ifo}$ are consistent with each other. In general, the body of a communication procedure can be any atomic action reading from or writing onto the local variables of the exporting action system. In this case, the body $bd(Get)$ of *Get* reads an item from a queue *q*: $q \in l_{\mathcal{F}ifo}$.

In *parallel composition* of action systems, $\mathcal{C}lient \parallel \mathcal{F}ifo$, the **execution** clause of the composed system is by definition:

$$\textbf{do } pop \;[]\; waitGet \textbf{ od}$$

The construct $pop \;[]\; waitGet$, where *pop* calls *Get* (**call** command) and *waitGet* awaits such a call (**await** command), is regarded as a single atomic action. Let us here name this action *popGet* and define it by:

$$popGet \mathrel{\widehat{=}} (pop_1; waitGet_1; bd(Get)[a/x]; waitGet_2; pop_2)$$

where $[a/x]$ substitutes the variable *a* that is used as an actual parameter, with the formal parameter *x*, and thus *Get* writes the head of the queue *q* onto *a*, which passes the requested value from $\mathcal{F}ifo$ to

Client. Hence, communication is based on sharing an action in which data is atomically passed from system to system by executing the body of a procedure.

3 SYSTEMC

SystemC (The OSCI, 2002) can be used in designing cycle-accurate models of software algorithms, hardware architectures and interfaces of SoC and system-level designs. It is a suitable tool for hardware modelling, even though it is built on a high-level, object-oriented software programming language. Being able to replace actual hardware description languages, such as VHDL and Verilog, in hardware design SystemC makes it possible to model entire HW/SW systems in a single description language. With SystemC designer can use all the object-oriented features and development tools of C++ (Stroustrup, 1997) in addition to the system architecture constructs, such as hardware timing, concurrency and reactive behaviour, which are not included in standard C++. Both software and hardware partitions that are written in SystemC can also be tested using the same test bench, provided by SystemC, without any need for language conversions between different abstraction levels. SystemC models can be refined down to lower abstraction levels ending up with an implementable design. The refinement process may be performed part by part so that one design may consist of system parts at several different abstraction levels. Nevertheless, the design remains simulatable.

A SystemC model consists of *modules* that break the design into several independent components. Modules implement data encapsulation by hiding local data and algorithms from the environment. Modules may contain also hierarchies of other modules. Modules' inner functionality is produced by running *processes* that are triggered by *events*. Processes of different modules communicate with each other via *channels* that are accessed through *ports* and *interfaces*. A port communicates with its designated channel through an interface, which introduces the methods that the port can use to access the channel. Channels are either *primitive channels* or *hierarchical channels* depending on their complexity. A primitive channel does not contain any other SystemC structures, whereas hierarchical channels may contain other modules and channels as well as internal processes. In practice, hierarchical channels may be as complex structures as modules.

The SystemC *scheduler* acts as a system kernel that controls the SystemC *simulation*. The scheduler handles the timing and order of the processes running in the system and updates the communication channels when needed. The scheduler catches and manages the events that processes raise. This way the scheduler acts as an intermediate in interprocess communications triggering processes by request of other processes. Concurrency of actions is modelled using the notion of delta delay. Simulation time remains stopped while the scheduler manages the events, process operations and channel updates of the simulation time point in question. Communications may be modelled in the simulation synchronously or asynchronously depending on the specification.

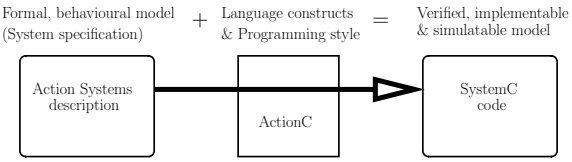

Figure 4: The concept of ActionC.

4 ACTIONC

ActionC can be considered as the implementation of the Action Systems formalism in the SystemC environment but ActionC is also a design method that follows the programming style and structure of Action Systems descriptions (Fig. 4). The objective of this merger is to utilise the best parts of both system modelling languages. Action Systems is a fine language in system specification and verification throughout a design process but it lacks the kind of commercial tools that would spread the use of the language beyond the borders of the academic world and promote the acceptance of the language in the industry. On the other hand, SystemC is a powerful tool in system model simulation, refinement and testing but it is short of a rigorous method that would ensure the correctness of the model in the initial system description and also preserve it during the refinement process. The idea of ActionC is to combine the formal features of Action Systems with the benefits of the SystemC environment, thus creating a framework for writing executable and refinable specifications that are verified in a formal manner.

The focal point of this paper is the ActionC implementation of the Action Systems type of *procedure based communication model*. In Action Systems, communication between systems is an essential issue when modelling complex embedded systems with multiple modules and processors. The procedure based communication model of Action Sys-

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

Action Systems		SystemC
action system	⇔	Module
in variable	⇔	sc_in⟨⟩
out variable	⇔	sc_out⟨⟩
inout variable	⇔	sc_inout⟨⟩
(local) **variable**	⇔	private C++/ SystemC variable
private procedure	⇔	private C/ C++ void method
public procedure	⇔	SystemC hierarchical channel
function	⇔	private C/C++ non-void method
action	⇔	Module member
execution loop	⇔	SystemC thread process

tems corresponds to the abstraction level of transaction level modelling, which is also the most usual modelling level in SystemC design. Therefore, communication modelling must be properly handled also in the ActionC framework. Before going further into the issues of inter-module communication, we will introduce the more obvious similarities between Action Systems and SystemC languages.

4.1 Matching SystemC and Action Systems Elements

SystemC and Action Systems share several language constructs as can be concluded from their short introductions in the previous sections. The directly mappable constructs are gathered in Table 1. Both languages use modularisation in creating local scopes: SystemC uses modules while an action system is the corresponding structure in the Action Systems formalism. In ActionC we call this structure an *ActionC module*. This structure also has an interface that it uses in the communication with the environment. The atomic actions of Action Systems are implemented as member functions of the ActionC module. The member functions may be either normal C++ methods or SystemC processes. These member functions are called *ActionC actions* and they are executed by performing *action calls*.

The **execution** loop of each of the Action Systems module 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

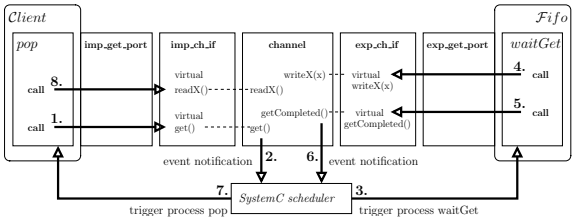

Figure 5: FIFO and its client communicating through a public procedure.

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. When the channel structure is more complex, a SystemC hierarchical channel is a more practical solution. For example, the procedure based communication model of Action Systems can be implemented in ActionC using the SystemC hierarchical channel. This implementation will be elaborated in Sect. 4.2.

4.2 Procedure based Communication

The possibility to hide detailed communication primitives at high abstraction levels allows a designer to concentrate on the functionality of the system under design. In Action Systems, as introduced in Sect. 2.2, the implementation of communication activities is hidden into a procedure based communication model. In the ActionC framework the procedure based communication is modelled by using a combination of SystemC's hierarchical channels, ports and interfaces at the transaction level of abstraction. With the combination of these artefacts we are able to confine the functionality of communication procedures and to have one channel per communication procedure. Ports and interfaces are used to connect a hierarchical channel to a system that exports it and to systems that call it, that is, in both ends of a hierarchical channel we have a port-interface couple. The functionality of an interface depends on its type: *exported* or *imported* communication channel. At the imported side the interface has a minimum amount of methods because the properties of the imported channel is introduced by a system that exports it. At the exported side the system has a wide-ranging set of methods to use the channel. Communication channels are introduced in and exported by a callee and imported by a caller, as in Action Systems. The implementation follows the blocking wait practice in which the calling action waits until the callee is ready to form the communication channel between the two modules.

The operation of the ActionC procedure based communication channel, in terms of the Action Systems communication procedure call, is illustrated in Fig. 5, which returns us to consider the FIFO structure we used as an action system example in Sect. 2. Again, action system $Client$ needs to extract data from the $Fifo$ using the public procedure *Get* exported by $Fifo$ and imported by $Client$. In general the data integrity during a 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 of the phases is determined by the existence of a return value. In this case the phases are: (1) The Action system $Client$ begins the communication procedure 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, which (3) triggers the process *waitGet*. The process *waitGet* is sensitive to the particular event. The concept of sensitivity in SystemC corresponds to the function of the **await** command in the Action Systems model. After being invoked (4) the process *waitGet* calls the *writeX()* method provided by its interface to write the first element of the queue into the channel's local variables. After the *writeX()* method has returned (5) process *waitGet* employs the channel interface method *getCompleted()*, which (6) sends an event notification to the SystemC scheduler. (7) As a result from the notification, the SystemC scheduler informs process *pop* of the completion of the communication procedure, that is, continues its execution from the point of last suspension. (8) By calling the interface method *readX()* process *pop* collects the value written to the channel by *waitGet*, thus finalising the communication procedure.

This implementation follows the atomicity concept of the procedure based communication model in Action Systems. The SystemC scheduler ensures that the entire procedure call from phase (1) to phase (8) is a continuous operation that is advanced and completed without interruptions.

5 CONCLUSIONS

We have taken the initial steps in creating a formal framework for modelling and verifying embedded computer systems with the SystemC environment. Using the presented *ActionC* framework designer is able to reliably simulate Action Systems descriptions. At the present phase of development *ActionC* still is more of a coding style than a fully implemented class library to be used with SystemC. In this paper we elaborated the ActionC implementation of the procedure based communication model used by the Action Systems formalism. The resulting model is a transaction level model of a communication method that uses remote function calls and blocking wait.

REFERENCES

Back, R.-J. (1988). A Calculus of Refinements for Program Derivations. *Acta Informatica*, 25:593–624.

Back, R.-J. and Kurki-Suonio, R. (1983). Decentralization of Process Nets with Centralized Control. In *PODC '83: Proc. of the second annual ACM symposium on Principles of Distributed Computing*, pages 131–142. ACM.

Back, R.-J. and von Wright, J. (1998). *Refinement Calculus: A Systematic Introduction*. Springer-Verlag.

Degerlund, F., Walden, M., and Sere, K. (2007). Implementation Issues Concerning the Action Systems Formalism. In *PDCAT '07: Proc. of the 8. Int. Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 471–479. IEEE.

Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice-Hall, Inc.

Habibi, A. and Tahar, S. (2005). Design for Verification of SystemC Transaction Level Models. In *Proceedings of the conference on Design, Automation and Test in Europe*, pages 560–565. IEEE.

Patel, H., Shukla, S., Mednick, E., and Nikhil, R. (2006). A Rule-Based Model of Computation for SystemC: Integrating SystemC and Bluespec for Co-Design. In *Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proc. 4. ACM and IEEE International Conference on*, pages 39–48. IEEE.

Sere, K. and Back, R.-J. (1994). From Action Systems to Modular Systems. In *FME'94: Industrial Benefit of Formal Methods*, pages 1–25. Springer-Verlag.

Stroustrup, B. (1997). *The C++ Programming Language*. Addison-Wesley, USA, 3. edition.

The OSCI (2002). *SystemC Version 2.0 User's Guide. Update for SystemC 2.0.1*.

The OSCI (2005). *IEEE Std 1666-2005, SystemC Language Reference Manual*. IEEE.

Vardi, M. Y. (2007). Formal Techniques for SystemC Verification. In *Proceedings of the 44th annual conference on Design automation*, pages 188–192. ACM.