speeds
EUROPE'S BEST IN SYSTEM DEVELOPMENT

Information Society Technologies

Project Number: 033471

SPEEDS

Speculative and Exploratory Design

in Systems Engineering

Integrated Project

Information Society Technologies

D2.4.6 Hosted Simulation – Proof of concepts for continuous systems

Due date of deliverable: T0+30
Actual submission date:

Start date of project: May 1st 2006 Duration: 48 months

Revision: in review

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

PROPRIETARY RIGHTS STATEMENT
THIS DOCUMENT CONTAINS INFORMATION, WHICH IS PROPRIETARY TO THE SPEEDS CONSORTIUM. NEITHER THIS DOCUMENT NOR THE INFORMATION CONTAINED HEREIN SHALL BE USED, DUPLICATED OR COMMUNICATED BY ANY

MEANS TO ANY THIRD PARTY, IN WHOLE OR IN PARTS, EXCEPT WITH THE PRIOR WRITTEN CONSENT OF THE SPEEDS CONSORTIUM THIS RESTRICTION LEGEND SHALL NOT BE ALTERED OR OBLITERATED ON OR FROM THIS DOCUMENT

Approvals

	Family Name	Name	Company	Date	Signature
SPEEDS Coordinator	Döhmen	Gert	AIG		
Quality Assurance	Döhmen Winokur	Gert Michael	AIG IAI		
Configuration Management	Döhmen	Gert	AIG		

Document History

Version	Date	Modification	Person
v.1.0.0	11/02/2007	First Version	Eldad Palachi
v.1.0.1	13/05/2007	Second Version	Eldad Palachi
v.2.0	13/01/2009	Third Version	Leonardo Mangeruca
			Gianluca Codella
			Alberto Ferrari
			Christos Sofronis
v3.0	12/02/2009	Fourth Version	Leonardo Mangeruca
			Alberto Ferrari
			Christos Sofronis
v4.0	30/03/2010	Fifth Version	Leonardo Mangeruca
			Alberto Ferrari
			Christos Sofronis
			Orlando Ferrante
		Version renamed to d.2.4.6	

LIST OF CONTENTS

1. OVERVIEW 6

2. TYPES OF INTERACTION OF AN HRC 7

3. COMPONENT INVOCATION, SCHEDULING AND TIME 8

4. EXPORTED HRC COMPONENT STRUCTURE 9

5. XMI DESCRIPTION OF AN HRC COMPONENT IN CONTEXT OF HOSTED SIMULATION 10

6. DATA TYPES 11

7. TIME REPRESENTATION 11

8. ROLE OF SPEEDS WRAPPERS 11

9. API FOR COMPONENTS EXPORTED BY A BMT TOOL 12

9.1 EXPORTING NON-PRIMITIVE DATA TYPES DEFINED IN THE BMT TOOL 12
9.2 PROVIDED SERVICES API 12
9.2.1 Instantiating a component 12
9.2.2 Getting the time resolution associated with a component 12
9.2.3 Setting an in (or inout) flow via a port 13
9.2.4 Getting an out (or inout) flow via a port 14
9.2.5 Signaling the presence of an event via an in\inout flow 14
9.2.6 Signaling the absence of an event via an in\inout flow 15
9.2.7 Doing a discrete computational step 15
9.2.8 Committing the discrete state after step 16
9.2.9 Evaluating the continuous time step 18
9.2.10 Doing a continuous time commit 18
9.2.11 Destroying a component 19
9.3 REQUIRED SERVICES API 20
9.3.1 Setting the presence of an event in a given simulation iteration 20
9.3.2 Setting the absence of an event in a given simulation iteration 20
9.3.3 Failure notifications 21

10. HOSTED SIMULATION PROTOCOL 22

11. SIMPLE EXAMPLE: TWO CONNECTED COMPONENTS 23

12. EXPORTED COMPOSITION FROM THE ACT TOOL 26

13. HST DATA EXCHANGE 26

14. REFERENCES 26

LIST OF FIGURES

FIGURE 1: WORKFLOW FOR EXPORTING, COMPOSING AND SIMULATING HRC COMPONENTS....... 7
FIGURE 2: RICH COMPONENT STRUCTURE (FIGURE 2.3 IN THE D.2.1.B SPEEDS META-MODEL [1]). 8
FIGURE 3: HRC EXPORTED DATA STRUCTURE 10
FIGURE 4: SIMPLE EXAMPLE - TWO CONNECTED HRC COMPONENTS (PARTS).................................. 24
FIGURE 5: INTERACTION BETWEEN THE COMPONENTS AND THE SIMULATOR 25

LIST OF TABLES

TABLE 1: PRIMITIVE TYPES MAPPING FROM SPEEDS TO C 11
TABLE 2: MAXACTIVATIONTIME AND IDLETIMEINTERVAL FOR SOME NOTEWORTHY CASES.... 17

ACRONYMS

- HRC: Heterogeneous Rich Component
- COTS: Commercial Of-The-Shelf
- BMT: Behavioural Modeling Tool
- ACT: Architectural\Compositional Tool
- HST: Hosted Simulation Tool

1. Overview

A common way to validate system models is by using simulation. In the case of SPEEDS, the system might be composed of HRCs designed in different tools, which poses a challenge on how to simulate the entire composed system.

The SPEEDS project takes a hosted simulation approach to resolve this challenge: in this approach the HRC components are exported to a standard format, and then imported to the composed system that can be simulated by a single simulation tool.

This approach has the following advantages compared with the co-simulation approach:

1. The user is not required to have all the tools to execute the simulation. Only the simulation tool is needed.
2. All the animated views are available in a single tool
3. Better performance: no overhead in using a message bus and coordination between different simulators
4. Ability to simulate interaction with an HRC component even if the design tool has no simulation capabilities

The main drawback of this approach is that one can only monitor the interactions between the components, but cannot simulate the internals of every component.

We have identified three roles that tools may play:

1. Behavioural Modelling Tool (BMT): Allows the specification of component behaviour and exports it as a black box component in the SPEEDS format. The tool should also provide an implementation of the component
2. Architectural\Composition Tool (ACT): A tool used to specify the composition of all the components and export it using SPEEDS meta-model
3. Hosted Simulation Tool (HST): A tool capable of importing the SPEEDS composition model and simulating it

It should be noted that a single tool may provide more than one function.

Figure 1 shows the canonical workflow users have to follow to use hosted simulation: first the components are designed in the BMT tools and are exported to the standard HRC format. Then the components are imported to the ACT tool which composes the system to be simulated. Next, the HST tool can read the composed system and simulate it.

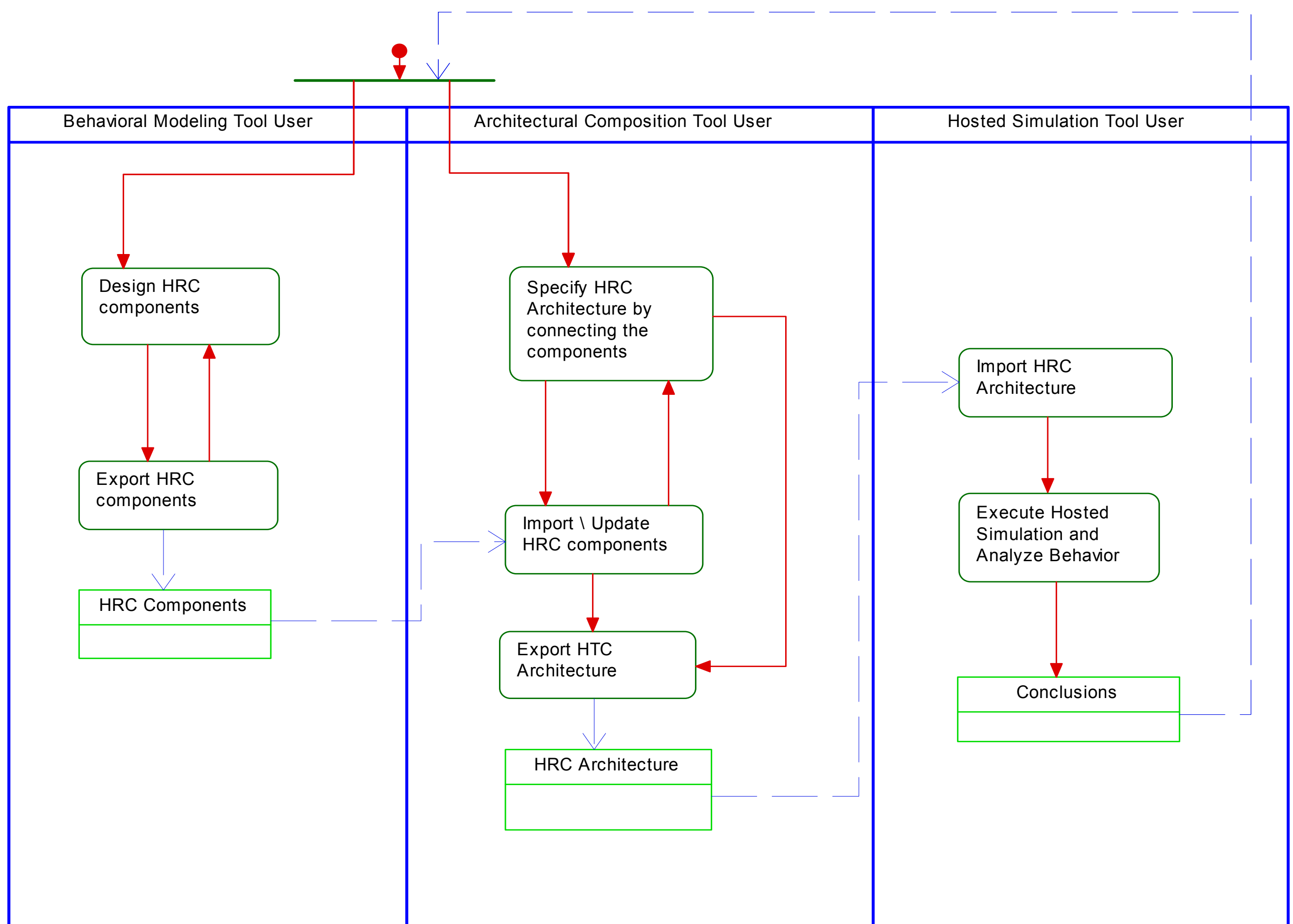

Figure 1: Workflow for exporting, composing and simulating HRC components

2. Types of interaction of an HRC

Figure 2 taken from reference [1] shows how the SPEEDS meta-model defines the structure of an HRC. Since hosted simulation treats the imported HRC as black box; we concentrate on the interaction points of the component: its ports. As seen in the meta-model, a port may have one or more flows (via its port specification type), where each flow is either in, out or inout and is of kind discrete, continuous or event.

In this version of the document we only deal with discrete components under the L0 HRC syntax. We do not deal with continuous time components nor with the L1 HRC syntax.

A discrete flow means that the data via the port is always accessible and evolves discretely. A continuous flow means that the data is always accessible and it evolves continuously (measuring the value at high rate may produce different values for each measurement). An event flow means an event is sent at a certain time via the port.

Based on this we identify the following interactions between rich components:

1. Exchanging a value from an out flow of one component to an in flow of another.
2. Sending a signal (event) from one component to another. The event is sent synchronously in L0.

It should also be noted that multiple writers on the same flow are not allowed.

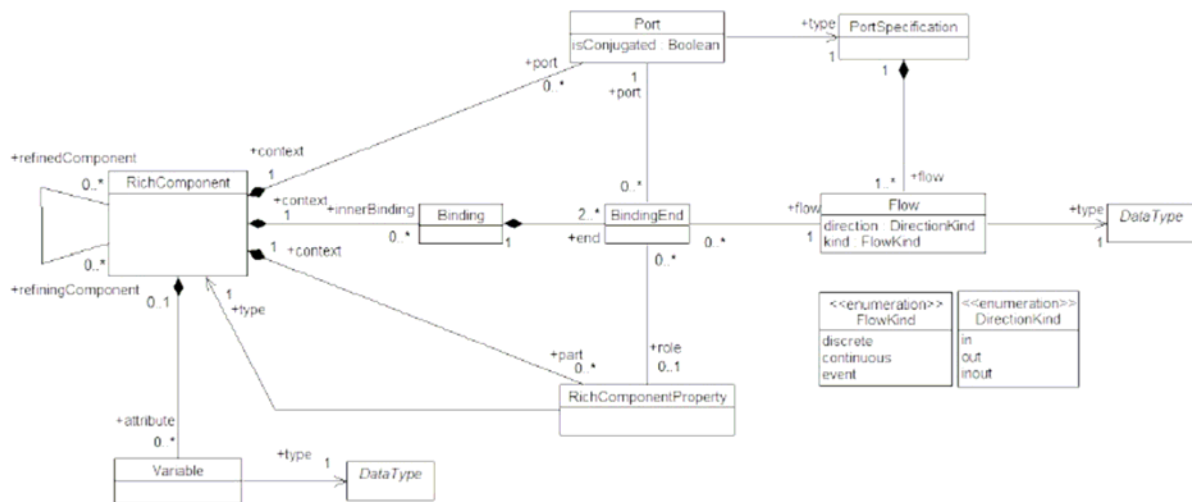

Figure 2: Rich Component structure (figure 2.3 in the D.2.1.b SPEEDS meta-model [1])

3. Component Invocation, Scheduling and Time

To support discrete HRC components and to avoid non-determinism due to invocation order by the simulation environment, during each simulation iteration the components are invoked in four phases:

- Discrete computation phase: Every component is triggered to perform a computational step until its computational task is complete. In this phase time is not advanced, components do not update their state and data is not exchanged between them, except for data that is relayed in the form of events: a component may emit an event at the end of a computational phase, in which case the event will be relayed to the connected components.
- Discrete commit phase: In the commit phase components update their state, data is exchanged between the connected components and, at the end of the phase, time advances. Every component sends to the hosting environment its idle time interval (during which it does not need to be triggered) and its max activation time, which is the longest period in which the component does not require to be activated. Based on these two constraints the hosting environment calculates an upper bound on the next iteration time interval as the minimum of the max activation time values returned by the discrete components.
- Continuous time step evaluation phase: The upper bound on the next iteration time interval computed at the end of the discrete commit phase is used as the maximum integration time horizon for the continuous time (CT) part of the hybrid components. In the current version of the protocol, only decoupled hybrid components can be supported, i.e. components that do not exchange or share continuous time variables. In the CT step evaluation phase the CT step API of the hybrid component is invoked. This API must evaluate whether any discrete transitions occur in the time interval represented by the upper bound on the next iteration time computed at the end of the commit phase. In case discrete transitions occur, the component sends the time interval between the current simulation time and the time of the earliest transition to the hosting environment as its max activation time that is set equal to its idle time interval. If no discrete transition occurs, the max activation time and the idle time interval are set equal to the upper bound on the next iteration time interval computed in the previous phase. During the continuous time step evaluation phase the hybrid component must not update its state variables. In this phase the CT step API of all hybrid components are invoked cyclically. At the end of

each cycle, the upper bound on the next iteration time interval is updated with the minimum of the max activation time values returned by the hybrid components. The cycle terminates when this value does not change, which also represents the new bound on the next iteration time interval. Before proceeding to the next phase, the hosting environment selects the next iteration time interval that must be not larger than the upper bound computed in this phase.

- Continuous time commit phase: The upper bound on the next iteration time interval computed at the end of the previous phase is used as the maximum integration time horizon for the continuous time (CT) part of the hybrid components. In the current version of the protocol, only decoupled hybrid components can be supported, i.e. components that do not exchange or share continuous time variables. In the CT commit phase the CT commit API of the hybrid component is invoked. This API must integrate the CT part of the hybrid component until the time horizon has been reached. The discrete transition of the hybrid component must not be taken in the current phase of the protocol, while it must be taken in the next step and commit phases of the protocol. Before proceeding to the next simulation iteration, the hosting environment selects which components to trigger at the next iteration, namely those components whose idle time is not strictly larger than the currently selected next iteration time.

Section 11 contains a sequence diagram for a simple case of two components that demonstrates how the API is used by the simulator to implement these phases.

4. Exported HRC Component Structure

The HRC component being exchanged between the tools has two parts (Figure 3):

1. An XMI description which describes the component based on the SPEEDS profile. This contains the interfaces and abstractions describing the component in the terms of the SPEEDS meta-model (profile)
2. An implementation of the component as generated by the COTS tool that exported the component. The implementation must be “wrapped” by the SPEEDS adaptor to support the SPEEDS API. In SPEEDS we use C as the implementation language. The component is either exported as a set of source files or as a compiled DLL accompanied with header files.

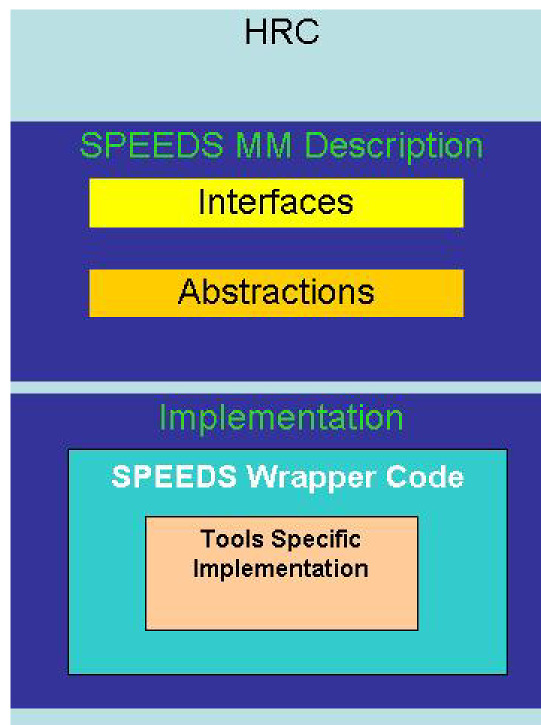

Figure 3: HRC Exported Data Structure

5. XMI Description of an HRC component in context of hosted simulation

The XMI of a component exported by a BMT tool must contain the HRC description of the component (Figure 2)

The relevant information is:

1. The name of the component (assumed to be unique in the name space of the composition)
2. List of parameters for initializing the component
3. The ports and their port specifications including all the flows
4. List of the HRC events that can be transmitted or received by the component
5. Complex data types defined and used in the component's interface
6. Signatures of the functions through which the components signal the presence or the absence of events to the environment that communicates it to connected components. These functions should be implemented by the hosted environment. See section: 9.3.
7. Failure Types – list of failures that may happen during the computation and commit phases. The hosting environment and the component may notify failure according to these types. See section 9.3.3.

The inner structure of the HRC component may not be exported since the simulation is black box. If it is exported, the HCT and ACT tool will ignore this information.

6. Data Types

The hosted simulation is implemented in C; hence a precise mapping between the SPEEDS Primitive Types (Boolean, Integer, Real, String and Void - see D2.1b) and C is specified in Table 1 below.

HRC Type	C Type
Boolean	bool, _Bool (ANSI C, C99 standard)
Integer	long
Real	double
String	char*
Void	void

Table 1: Primitive types mapping from SPEEDS to C

The non-primitive types: Array, Enumeration and Record should be translated to C user defined types (typedef). This is to allow exchange of non-primitive data between components designed in different tools. From a workflow perspective, one tool should specify the non-primitive type (in terms of primitive types) and then these types should be exported to other tools using XMI.

7. Time Representation

Time is represented as a 64-bit unsigned integer number. This number is to be interpreted by the hosting tool according to the chosen time resolution. The time resolution can be one of the following time units: “fs” (femtoseconds), “ps” (picoseconds), “ns” (nanoseconds), “us” (microseconds), “ms” (milliseconds) or “s” (seconds). Each component in the hosted simulation may specify its own time resolution by specifying an optional hosted simulation API (see Section 9.2.2). If this API is not provided by the component, a time resolution will be assigned to the component arbitrarily by the hosting simulator. For the simulation to be correct only untimed components are allowed to omit their time resolution API.

The simulator will represent time associating a time unit to a 64-bit unsigned integer number and this unit will not be greater than the smallest time resolution specified by any component.

8. Role of SPEEDS wrappers

C implementations coming from COTS tools may be “wrapped” in order to support SPEEDS API and to be able to represent as *undefined* the presence of flows of event kind in a given simulation iteration. As for the second purpose, it can be achieved by associating each flow of event kind with a boolean flag, which indicates whether the event presence is *undefined* (if flag is false) or *defined* (i.e. the event is definitely present or absent).

Whenever a component is triggered to perform a computational step, the wrapper is supposed to represent as *undefined* the presence of all its output events which depend directly at least on one input event whose presence is *undefined*, whereas the output events which depend directly only on input events whose presence is defined (or do not depend directly on input events at all) must be set present or absent, according to the results of the computation performed[1].

[1] It is worth noting that generally these dependencies are not static, but can vary dynamically (depending on input and state current values).

It should be noted that, as a consequence of this approach, both presence and absence of events must be communicated during the computational step (see Sections 9.2.5 and 9.2.6).

9. API for components exported by a BMT tool

The API functions (C signatures) are derived from the XMI description of the HRC component (exported from the BMT tool)

9.1 Exporting non-primitive data types defined in the BMT tool

The complex data types are only exported via the XMI and not as part of the C code. The ACT tool may generate the necessary C code for these types (note that all the types must have an explicit typedef).

When the ACT tool gets a non-primitive type from the BMT tool, it searches if a type with the same name is already defined in its repository. It there is no such type then the ACT tool creates a new type; if this type already exists the tool checks if the types match (have the same fields or elements). If there is no match then the user is either notified and\or given the option to override the existing type or skip the import.

9.2 Provided services API

The provided services are a set of C functions implemented by the HRC component for the simulator (the HST tool) to invoke.

9.2.1 Instantiating a component

Syntax:

Return type	Function name	Parameters
*<ComponentType>**	*<ComponentType>_Init*	*([parameters values list]);*

Parameters list: all the parameters required by the component to be instantiated and initialized.

Semantics:

This API is used to instantiate components.
It is assumed that all components are instantiated before execution begins (e.g. in T=0 all the components are already created)

Example:

To instantiate component A with initializer with parameters p1:Integer and p2:String one needs to invoke the C function:

A A_Init(long p1, char* p2);*

9.2.2 Getting the time resolution associated with a component

NOTE: This API is still to be accepted by the partners involved in the hosted simulation workpackage.

This API must be implemented by the component and is optional. If this API is present, the preprocessor directive "#define <ComponentType>_TIME_UNIT" must be present in the .h file of the component, where <ComponentType> is the component's type name written as it is, without any character case modifications.

Syntax:

Return type	Function name	Parameters
short	*<ComponentType>_TimeUnit*	*None*

This API has no parameters.
Return value: an integer value between 1 and 6.

Semantics:

This API is used to get the time resolution associated to a component. The time resolution is expressed in terms of one of the following time units: "fs" (femtoseconds), "ps" (picoseconds), "ns" (nanoseconds), "us" (micro-seconds), "ms" (milliseconds), "s" (seconds). The above time units are coded into the integer values 1 ("fs"), 2 ("ps"), 3 ("ns"), 4 ("us"), 5 ("ms"), 6 ("s").

Example:

Assuming that the simulation environment is using nanoseconds as the basic time unit for the simulation, to convert the time resolution of component MyComp to nanoseconds one needs to invoke the API as follows:

```
Unsigned long MyComp_time_resolution = 1; /* by default, resolution is equal to simulation resolution */

#ifdef MyComp_TIME_UNIT

        if ( MyComp_TimeUnit() >= 3 )
                MyComp_time_resolution=(MyComponent_TimeUnit() - 3)*1000;
        else
                hostedSimulationFailure("Hosted simulation resolution is too low");

#endif
```

When invoking the component step and commit functions (see Sections 9.2.7 and 9.2.8) the "*currentTime*" parameter must be divided by "*MyComp_time_resolution",* while when using the parameters *"idleTimeInterval"* and *"maxTimeInterval*" returned by the component's commit function, the simulation environment must multiply them by "*MyComp_time_resolution"*.

9.2.3 Setting an in (or inout) flow via a port

Syntax:

Return type	Function name	Parameters
Void	*<ComponentType>_Set_<PortName>_<FlowName>*	*(<ComponentType>* theInstance, const <FlowType>[*] value);*

<ComponentType> theInstance*: reference to the component
<FlowType> value: the value to be assigned; for Arrays and Records a pointer is used (i.e. <FlowType>* value)

Semantics:

The value passed as a parameter is assigned to the specified flow owned by the referenced component (data types must match).

Examples:

To set the value of flow X1:Integer via port P1 of MyComponent use the API function:
void MyComponent_Set_P1_X1(MyComponent theInstance, const long value);*

For a flow XList:MyList (where MyList is an Array) the API is:
void MyComponent_Set_P1_XList(MyCompoennt theInstance, const MyList* value);*

9.2.4 Getting an out (or inout) flow via a port

Syntax:

Return type	Function name	Parameters
<FlowType>[*]>	*<ComponentType>_Get_<PortName>_<FlowName>*	*(<ComponentType> * theInstance);*

<ComponentType> theInstance*: reference to the component
Return value: the current flow value; for Arrays and Records a pointer is retuned (i.e. <FlowType>*)

Semantics:

The current value of the specified flow owned by the referenced component is returned.

Examples:

To get the value of flow X1:Integer via port P1 of MyComponent use the API function:
long MyComponent_Get_P1_X1(MyComponent theInstance);*

For a flow XList:MyList (where MyList is an Array) the API is:
MyList MyComponent_Get_P1_XList(MyComponent* theInstance);*

9.2.5 Signaling the presence of an event via an in\inout flow

Syntax:

Return type	Function name	Parameters
void	*<ComponentType>_<PortName>_<FlowName>_SignalPresence*	*(<ComponentType> * receiver, const <EventDataType> value);*

<ComponentType> receiver*: reference to the receiver component
<EventDataType> value: value carried by the event; this parameter must be consequently omitted in case of event carrying no value.

Semantics:

The presence and the carried value (if any) of the specified event are communicated to the referenced receiver component.

If the emission of events (in the sense of signaling their presence) is intended to be used to debug purposes as a breakpoint for the simulator, it is advisable to stop the simulation at the end of the computation phase, but before the commit phase.

Example:

To signal the presence of the event MyEvent (carrying an integer value) to MyComponent via port P1 the API is:

void MyComponent_P1_MyEvent_SignalPresence(MyComponent receiver, const long value);*

9.2.6 Signaling the absence of an event via an in\inout flow

Syntax:

Return type	Function name	Parameters
void	*<ComponentType>_<PortName>_<FlowName>_SignalAbsence*	*(<ComponentType>* receiver);*

<ComponentType> receiver*: reference to the receiver component

Semantics:

The absence of the specified event is communicated to the referenced receiver component.
No value must be communicated when an event is absent.

Example:

To signal the absence of the event MyEvent to MyComponent via port P1 the API is:

void MyComponent_P1_MyEvent_SignalAbsence(MyComponent receiver);*

9.2.7 Doing a discrete computational step

Syntax:

Return type	Function name	Parameters
bool	*<ComponentType>_Step*	*(<ComponentType*> theInstance, const time currentTime);*

<ComponentType> theInstance*: reference to the component

time currentTime: simulation time value during the commit phase
Return value: (see below)

Semantics:

Every call to the step function invokes the referenced component to process its input data and perform all the required computations.

This function returns *false* if the presence (or absence) of some output event has become defined during the last computational step, otherwise it returns *true*.

Time does not progress during the step function execution and discrete data flow values are not communicated to other components. Ideally, components do not update their state in the step function. More precisely, the step function must not update the component's state if:

1. The component's input events are not all defined yet;
2. The component's output events are already all defined (i.e. if the component has already executed);
3. For timed components, the instant of time is not the one in which the component is expecting to be executed.

Moreover, when the component's input events become all defined, the next step function execution must define the presence/absence of all its output events and the value of all its present output events.

The above rules are necessary, because the hosted simulation protocol (see Section 10) is allowed to call the step function during a single computation phase an arbitrary number of times, in order to resolve the causal chain of events between the components.

At end of every step function execution the presence (or absence) of some output events may become defined. As a result of this, the component must communicate their presence (or absence) and this information will be conveyed to all connected components (sections 9.3.1 and 9.3.2). These components will store the present events and process them when the HST environment will call their step function.

Example:

For component MyComponent the API function is:
bool MyComponent_Step(MyComponent theInstance, const time currentTime);*

9.2.8 Committing the discrete state after step

Syntax:

Return type	Function name	Parameters
void	*<ComponentType>_Commit*	*(<ComponentType>* theInstance, const time currentTime, time* idleTimeInterval, time* maxActivationTime);*

<ComponentType> theInstance*: reference to the component
time currentTime: simulation time value during the commit phase

time idleTimeInterval*: (see below)
time maxActivationTime*: (see below)

Semantics:

The commit function causes the component to update its state and its discrete output (and inout) flow values. Once the commit phase is done for all components, the environment uses the get and set APIs (sections 9.2.2 and 9.2.4) to update the discrete inputs before starting a new computational phase using the step functions (section 9.2.7)

The currentTime parameter is an input from the hosting simulation environment to the component notifying it the current simulation time.

The *idleTimeInterval* parameter is an output parameter notifying the simulation environment that the component needs not be invoked during this time interval, relative to the currentTime.

The *maxActivationTime* parameter is an output parameter indicating that the component must be invoked at least once within this time interval, relative to the currentTime value. Consequently, if a component needs to be executed again at the same instant of simulation time, it should specify maxActivationTime = 0; if it needs to be executed within time T, it should specify maxActivationTime = T; finally, if it does not need to put any constraint on the next simulation iteration time, it should assign to maxActivationTime the largest representable time value (i.e. MAX_TIME = 2^{64}-1 seconds).

For the purpose of further clarification, the following common cases are explicitly illustrated: if a component is periodic (i.e. it is activated periodically, it is not triggered by events, it has no combinational input-output path), it should specify maxActivationTime = T and idleTimeInterval = T (where T is the period); if a component is event-driven (i.e. it is activated by some input events), it should specify maxActivationTime = MAX_TIME and idleTimeInterval = 0 (see Table 2).

COMPONENT	*maxActivationTime*	*idleTimeInterval*
Periodic	T	T
Event-driven	MAT_TIME	0

Table 2: maxActivationTime and idleTimeInterval for some noteworthy cases

Based on the maxActivationTime intervals the hosted simulation environment calculates the next simulation iteration time value and based on the idleTimeIntervals it determines the set of components which need to be invoked at that time. Then the simulation environment starts a cycle of step function invocations and subsequently invokes the commit function on the same set of components.

After the commit phase is completed, each component must reset as *undefined* the presence of all its input events.

Example:

For component MyComponent the API function is:

void MyComponent_Commit(MyComponent theInstance, const time currentTime, time* idleTimeInterval, time* maxActivationTime);*

9.2.9 Evaluating the continuous time step

This API must be implemented by the component and is optional. If this API is present, the preprocessor directive "#define <ComponentType>_CONTINUOUS_TIME_STEP" must be present in the .h file of the component, where <ComponentType> is the component's type name written as it is, without any character case modifications.

Syntax:

Return type	Function name	Parameters
void	*<ComponentType>_ContinuousTimeStep*	*(<ComponentType>* theInstance, const time currentTime, const time nextTime, time* idleTimeInterval, time* maxActivationTime);*

<ComponentType> theInstance*: reference to the component
time currentTime: current simulation time value
time nextTimeInterval: next simulation time interval, relative to the currentTime, computed after the discrete commit phase
time idleTimeInterval*: as in the commit phase API
time maxActivationTime*: as in the commit phase API

Semantics:

The continuous time step function is used to handle hybrid systems. The current version of the hosted simulation protocol is able to handle only decoupled hybrid components. i.e. components that do not exchange or share continuous time variables. This limitation is due to the fact that the current version of the protocol integrates components in the discrete time domain, while hybrid components must in general be integrated in the continuous time domain (the integration of components in the continuous and discrete time domains is not equivalent in general). For efficiency reasons it is also assumed that no combinational path is present between an output and an input of the hybrid or purely continuous time component.

The continuous time step function is used to refine the next simulation time step. After the discrete commit phase, the simulation protocol computes an upper bound on the next simulation step (see Section 3). This value is passed as the nextTime argument to the continuous time step functions. This function evaluates whether discrete transitions of the corresponding hybrid component occur between currentTime and currentTime+nextTimeInterval. If this is the case, let discreteTransitionTime denote the time of the earliest transition. Then, the component sets the maxActivationTime and idleTimeInterval to the value discreteTransitionTime–currentTime, otherwise they are set to the nextTimeInterval value.

The continuous time step function must not update the internal state of the corresponding component and must not emit any event.

9.2.10 Doing a continuous time commit

This API must be implemented by the component and is optional. If this API is present, the preprocessor directive "#define <ComponentType>_CONTINUOUS_TIME_COMMIT" must

be present in the .h file of the component, where <ComponentType> is the component's type name written as it is, without any character case modifications.

Syntax:

Return type	Function name	Parameters
void	*<ComponentType>_ContinuousTimeCommit*	*(<ComponentType>* theInstance, const time currentTime, const time nextTime, time* idleTimeInterval, time* maxActivationTime);*

<ComponentType> theInstance*: reference to the component
time currentTime: current simulation time value
time nextTimeInterval: next simulation time interval, relative to the currentTime, computed after the discrete commit phase
time idleTimeInterval*: as in the commit phase API
time maxActivationTime*: as in the commit phase API

Semantics:

The continuous time commit function is used to handle hybrid systems and purely continuous time components. The current version of the hosted simulation protocol is able to handle only decoupled hybrid components. i.e. components that do not exchange or share continuous time variables. This limitation is due to the fact that the current version of the protocol integrates components in the discrete time domain, while hybrid and purely continuous time components must in general be integrated in the continuous time domain (the integration of components in the continuous and discrete time domains is not equivalent in general). For efficiency reasons it is also assumed that no combinational path is present between an output and an input of the timed automaton, hybrid or purely continuous time component.

The continuous time commit function is called after the continuous step evaluation phase. After the continuous step evaluation phase, the hosted simulation protocol computes the next simulation time step, which is passed to the continuous time commit API together with the current simulation time. When called, the continuous time commit function integrates the hybrid component until the next simulation time step is reached. It is mandatory that no discrete transition occurs in this interval. This condition must be guaranteed by the maxActivationTime value returned by the corresponding continuousTimeStep function. The continuous time commit function can only update the continuous time state variables, it must not execute any discrete transitions, i.e. it must not update the discrete state variables of hybrid components nor emit any event, which instead will be dealt with in the next simulation iteration.

Upon completion of all the continuous time commit functions, the simulation time is updated and the next simulation cycle can start.

Example:

For component MyComponent the API function is:

void MyComponent_ContinuousTimeStep(MyComponent theInstance, const time currentTime, const time nextTime, time* idleTimeInterval, time* maxActivationTime);*

9.2.11 Destroying a component

Syntax:

Return type	Function name	Parameters
void	*<ComponentType>_Destroy*	*(<ComponentType>* theInstance);*

<ComponentType> theInstance*: reference to the component

Semantics:

This API cleans up all allocated memory by the component, the component is considered to be deleted after calling this API.

Example:

To destroy MyComponent the API is:

void MyComponent_destroy(MyComponent theInstance);*

9.3 Required services API

9.3.1 Setting the presence of an event in a given simulation iteration

Syntax:

Return type	Function name	Parameters
void	*<ComponentType>_<PortName>_<FlowName>_SetPresence*	*(<ComponentType>* emitter, const <EventDataType> value);*

<ComponentType> emitter*: reference to the emitter component
<EventDataType> value: value carried by the event; this parameter must be consequently omitted in case of event carrying no value.

Semantics:

Once invoked this API, the hosted simulation environment communicates the presence of the specified event to the connected components using the API in section 9.2.5.

Example:

To notify the emission of the event MyEvent (carrying an integer value) by MyComponent via port P1 to the hosted simulation environment the API is:

*void MyComponent_Port1_MyEvent_SetPresence(MyComponent*emitter, const long value)*

9.3.2 Setting the absence of an event in a given simulation iteration

Syntax:

Return type	Function name	Parameters
void	*<ComponentType>_<PortName>_<FlowName> _SetAbsence*	*(<ComponentType>* emitter);*

<ComponentType> emitter*: reference to the emitter component

Semantics:

Once invoked this API, the hosted simulation environment communicates the absence of the event to the connected components using the API in section 9.2.6.
No value must be communicated when an event is absent.

Example:

To notify the absence of the event MyEvent on port P1 of MyComponent to the hosted simulation environment the API is:
void MyComponent_Port1_ MyEvent_SetAbsence(MyComponent emitter);*

9.3.3 Failure notifications

This API is implemented by the environment to notify failures.
Syntax:

Return type	Function name	Parameters
void	*notifyFailure*	*(FailureType failure, char* message);*

failure: failure type identifier
message: failure message

Semantics:

The following failure type identifiers are predefined:

OutOfSynchLocalTime: local time of hybrid or continuous time component is different from current simulation time during step computation phase or commit phase or at the beginning of the continuous time step function.
ComponentInvocationUnexpected: by specification, the hosted simulation protocol can arbitrarily invoke the component's API. To satisfy this requirement each component is structured with a hosted simulation wrapper and a component's implementation. The hosted simulation wrapper filters out unexpected component's invocations. For example, for a periodic component, the component's wrapper filters out component's invocations at times that are not multiples of the component's period. Moreover, for component's that require a single invocation to the step API, the component's wrapper filters out all the step API invocations but one, as defined in the step API specification. This failure type is useful to notify a failure in this wrapper filtering operation.

ComponentInvocationMissing: this failure type is used to notify an error either in the protocol or in the component's wrapper. For example, if a periodic component misses one period or if a component's commit function is not invoked.

Unknown: this failure type is used for failures that do not fall in any of the previous cases.

10. Hosted simulation protocol

The hosted simulation protocol is substantially composed of two levels of nested cycles. Each iteration of the outer cycle represents a simulation iteration and consists of four phases: a discrete computation phase, a discrete commit phase, a continuous step evaluation phase and a continuous time commit phase, as described in Section 3. At the end of the continuous time step evaluation, the next simulation time value is determined. The simulation time is advanced accordingly at the end of the continuous time commit phase. There are two inner cycles corresponding to the discrete computation phase and the continuous time step evaluation phase, which may generally require more than one single iteration, as described below.

Discrete computation phase (first inner cycle): before the beginning of this phase, the presence of all events is represented as *undefined* and a global boolean flag is set to *false*. When an iteration begins, the flag is set to *true* and the cycle does not end until the flag is detected to be *true* at the end of some iteration. During each iteration, the step function is invoked on all components in arbitrary order. The step invocation is allowed to be skipped for those components whose idleTimeInterval value from the last simulation iteration is greater than the last simulation time increment. Whenever one of the step functions returns *false*, the global flag must be set to *false* as well, so that another inner cycle iteration is required.

Provided that the absence of events is explicitly communicated (through the appropriate API, see Section 9.2.6) and the wrappers operate as specified in Section 8, the computation phase is guaranteed to terminate in a finite number of iterations. The number of iterations required is bounded by N+1, where N is the number of components. At the end of the computation phase all the flows of event kind are guaranteed to be defined (i.e. either present or absent), unless there exist combinational loops in the system.

Combinational loops are caused by the component's input/output direct dependencies, also called combinational paths, i.e. an output event whose value depends on the value of some inputs at the same simulation iteration.
Combinational paths are in general dynamic, since they may depend on the value of the component's inputs. Nonetheless static analysis can identify potential input/output direct dependencies. Potential combinational paths of different components can be interconnected into a loop, forming a potential combinational loop.

If input/output direct dependencies are checked statically and coded in the component's hosted simulation wrapper, the protocol cannot simulate potential combinational loops, but they can be dynamically detected (without the need for topological analysis, because in such a case all the events in the loop will remain *undefined*). Similarly, if input/output direct dependencies are checked dynamically by the component's hosted simulation wrapper, the protocol cannot simulation combinational loop, but it is able to distinguish between real and potential combinational loops, being therefore capable of simulating potential combinational loops that are not real. Real combinational loops can still be detected, since all the events in the loop will remain *undefined.*

Discrete commit phase: As described in Section 3, after the discrete computation phase, the discrete commit phase is carried out (the commit function is invoked once on all components in any order, optionally except those which do not need to execute based on their idle time interval). After the commit phase, an upper bound on the next simulation step is computed for the discrete part as the minimum of the maximum activation time values returned by the commit

APIs. The bound on the next simulation step is passed to the continuous time step evaluation phase.

Continuous time step evaluation phase (second inner cycle): The upper bound on the next simulation step is refined during this phase. During each iteration, the continuous time step function is invoked on all components in arbitrary order. The step invocation is allowed to be skipped for those components whose idleTimeInterval value from the last simulation iteration is greater than the last simulation time increment. Each continuous time step function evaluates whether any discrete transitions occur within the upper bound on the next iteration time step computed in the previous phase. If this is the case, the function returns the interval between the current simulation time and the time of the earliest discrete transition as its maxActivationTime. Otherwise, the function returns the upper bound on the next simulation step as its maxActivationTime. The idleTimeInterval can be set equal to the maxActivationTime value. At the end of each iteration, the upper bound on the next simulation step is updated with the minimum among the maxActivationTime values returned by the continuous time step functions. The cycle terminates when such an upper bound is no further changed. Since the upper bound is positive and never increased and the measure of time is quantized, the cycle is guaranteed to terminate in a finite number of iterations.

Continuous time commit phase: At the end of the previous phase, the simulation protocols selects the next simulation time step, which must be not larger than the upper bound computed in the previous phase. In this phase the continuous time variables of hybrid components is updated by integrating the continuous time part up to the next simulation time. The discrete state variables of the hybrid components are not updated in this phase, while they will be handled in the next simulation iteration. At the end of this phase, the current simulation iteration terminates, the simulation time can be advanced and the next simulation iteration can start.

11. Simple Example: Two Connected Components

To clarify the usage of the API we consider a simple case of two connected HRC components (parts) as shown in Figure 4. The port p:PS is typed by the port specification PS that has three flows: an x of type int which is discrete and with direction out, and two event flows of type int with direction out, evt1 and evt2 respectively. The port p of component A is typed by PS and so is port p of component B, however this port is conjugated.

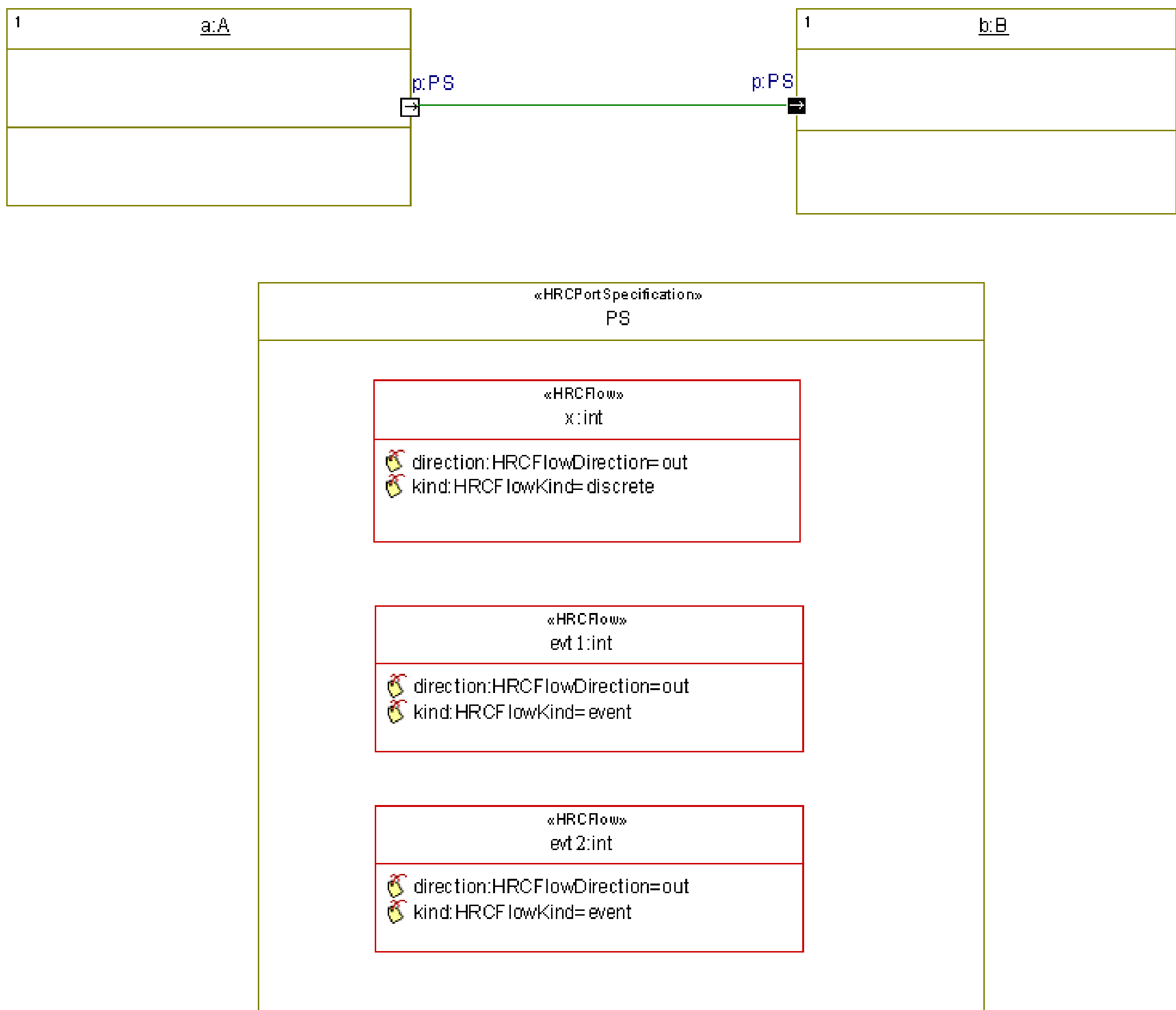

Figure 4: Simple Example - Two connected HRC components (parts)

Figure 5 is a sequence diagram showing how the simulator should use the API to trigger and monitor the components. This demonstrates the general principles of the API usage.

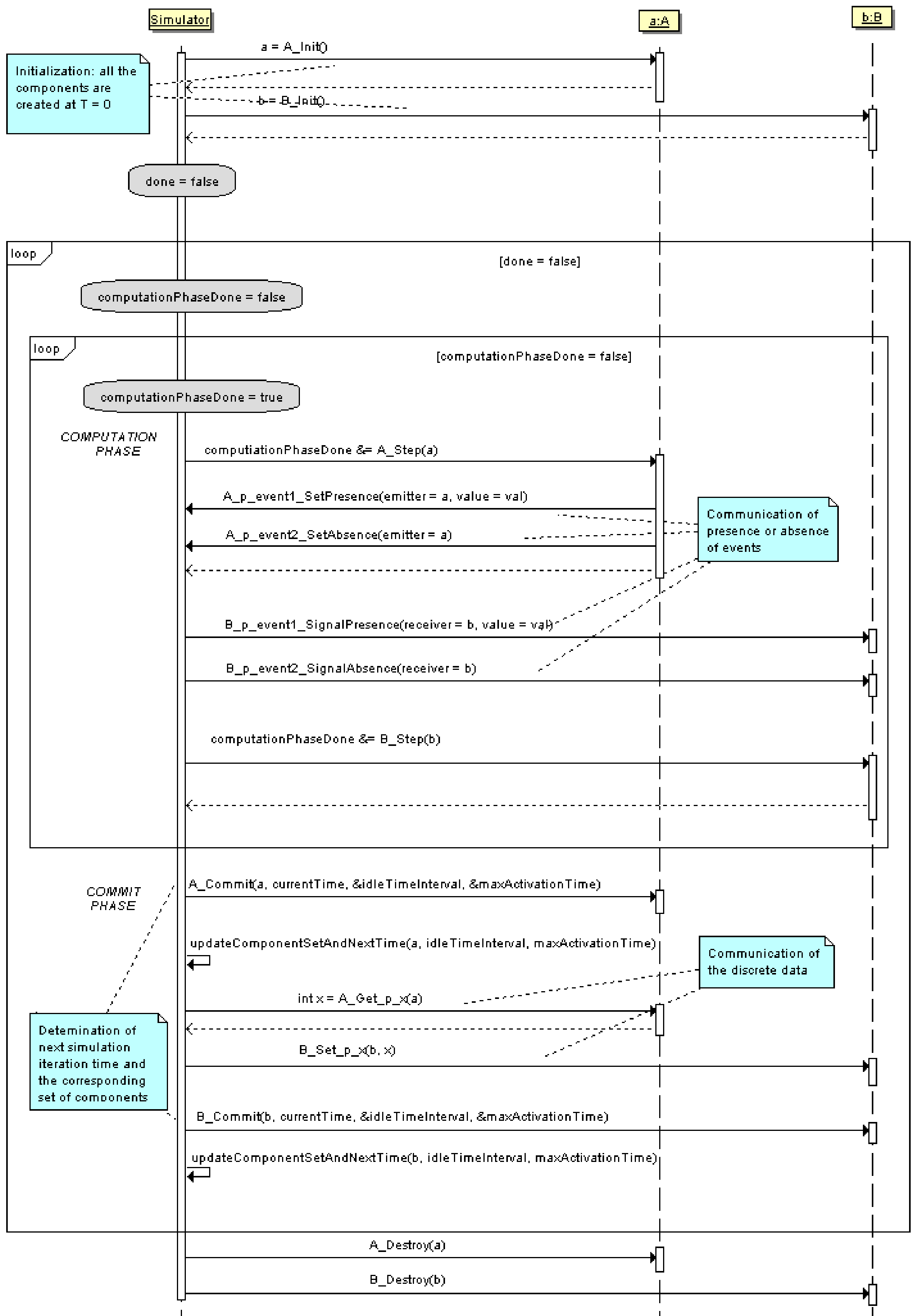

Figure 5: Interaction between the components and the simulator

12. Exported Composition from the ACT tool

The Architecture Composition Tool (ACT) exports the composed system to the Hosted Simulation Tool (HST) using XMI. The XMI should include:

1. All the XMI information as exported by the BMT tools. This can be implemented by pointing to the exported XMI as generated from the BMT tool, or by duplicating the data in the XMI exported by the ACT tool. The advantage of duplicating the data is that in some cases the composition is exported from the ACT tool before the BMT tool exports its component (i.e. it is just a stub)
2. A list of the C source files implementing the BMT components and their mapping to the component's in the model

13. HST Data exchange

The HST tool should be able to import the composed system from the ACT and the C source files and generate the implementation of the composed system so it can be simulated. This implementation can be in any language, compiled or interpreted.

The HST is not required to export any modelling data back to the ACT or BMT tools. The results of the simulation are analyzed based on the specific HST tool features.

Having a standard export of the simulation results to analysis tools or alternatively an API to trace and control the simulation by an analysis tool is currently outside the scope of the project.

14. References

1. D.2.1.b SPEEDS Meta-model Syntax and Static Semantics

2. Stephan A. Edwards, Edward A. Lee, "The semantics and execution of a synchronous block-diagram language", Science of computer programming 48 (2003) 21-42