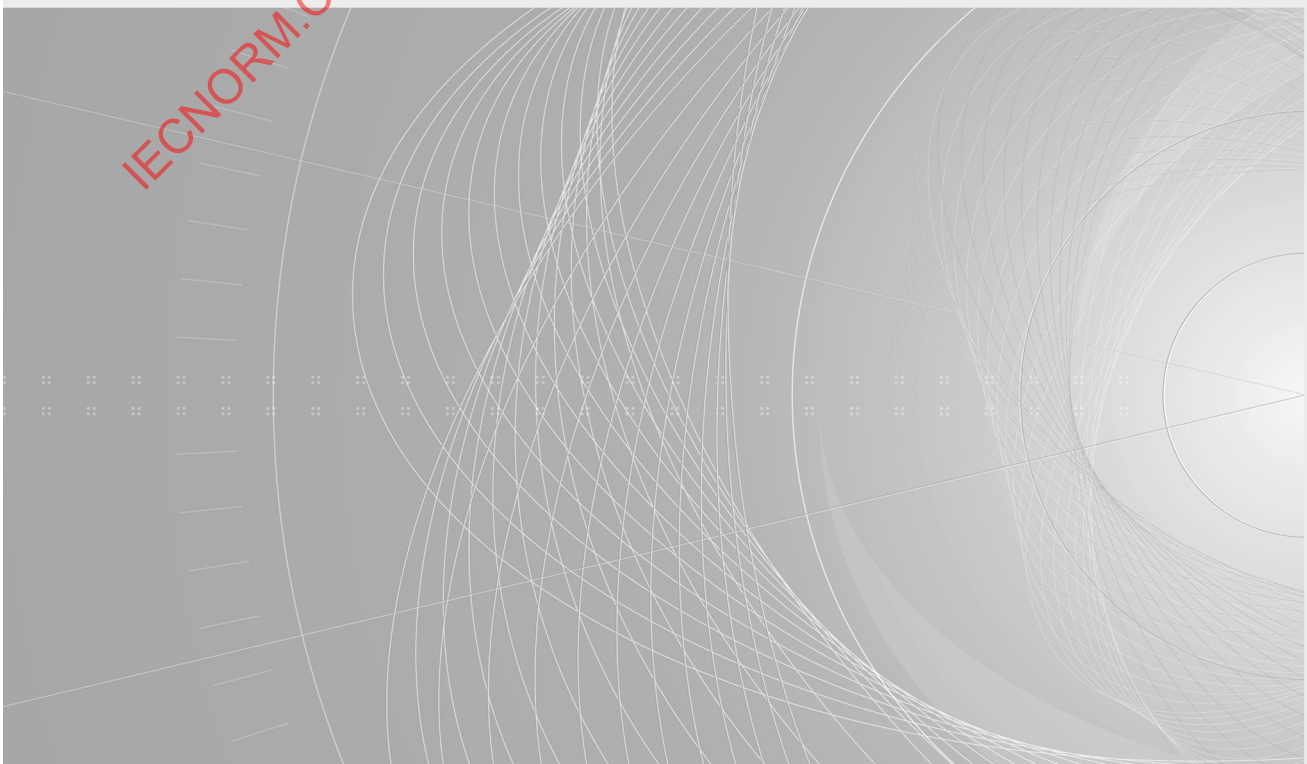
IEC 62541-10

Edition 3.0 2020-07
REDLINE VERSION

INTERNATIONAL STANDARD

colour inside

OPC unified architecture – Part 10: Programs



IEC 62541-10:2020-07 RLV(en)

IEC Central Office
3, rue de Varembé
CH-1211 Geneva 20
Switzerland

Tel.: +41 22 919 02 11
info@iec.ch
www.iec.ch

About the IEC
The International Electrotechnical Commission (IEC) is the leading global organization that prepares and publishes International Standards for all electrical, electronic and related technologies.

About IEC publications
The technical content of IEC publications is kept under constant review by the IEC. Please make sure that you have the latest edition, a corrigendum or an amendment might have been published.

IEC publications search - webstore.iec.ch/advsearchform
The advanced search enables to find IEC publications by a variety of criteria (reference number, text, technical committee,…). It also gives information on projects, replaced and withdrawn publications.

IEC Just Published - webstore.iec.ch/justpublished
Stay up to date on all new IEC publications. Just Published details all new publications released. Available online and once a month by email.

IEC Customer Service Centre - webstore.iec.ch/csc
If you wish to give us your feedback on this publication or need further assistance, please contact the Customer Service Centre: sales@iec.ch.

Electropedia - www.electropedia.org
The world's leading online dictionary on electrotechnology, containing more than 22 000 terminological entries in English and French, with equivalent terms in 16 additional languages. Also known as the International Electrotechnical Vocabulary (IEV) online.

IEC Glossary - std.iec.ch/glossary
67 000 electrotechnical terminology entries in English and French extracted from the Terms and Definitions clause of IEC publications issued since 2002. Some entries have been collected from earlier publications of IEC TC 37, 77, 86 and CISPR.

IEC 62541-10

Edition 3.0 2020-07
REDLINE VERSION

INTERNATIONAL STANDARD

colour inside

OPC unified architecture – Part 10: Programs

INTERNATIONAL
ELECTROTECHNICAL
COMMISSION

ICS 25.040.40; 35.100.05

ISBN 978-2-8322-8638-8

CONTENTS

INTERNATIONAL ELECTROTECHNICAL COMMISSION

_____

OPC UNIFIED ARCHITECTURE –

Part 10: Programs

FOREWORD

1) The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides (hereafter referred to as "IEC Publication(s)"). Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation. IEC collaborates closely with the International Organization for Standardization (ISO) in accordance with conditions determined by agreement between the two organizations.
2) The formal decisions or agreements of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees.
3) IEC Publications have the form of recommendations for international use and are accepted by IEC National Committees in that sense. While all reasonable efforts are made to ensure that the technical content of IEC Publications is accurate, IEC cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.
4) In order to promote international uniformity, IEC National Committees undertake to apply IEC Publications transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC Publication and the corresponding national or regional publication shall be clearly indicated in the latter.
5) IEC itself does not provide any attestation of conformity. Independent certification bodies provide conformity assessment services and, in some areas, access to IEC marks of conformity. IEC is not responsible for any services carried out by independent certification bodies.
6) All users should ensure that they have the latest edition of this publication.
7) No liability shall attach to IEC or its directors, employees, servants or agents including individual experts and members of its technical committees and IEC National Committees for any personal injury, property damage or other damage of any nature whatsoever, whether direct or indirect, or for costs (including legal fees) and expenses arising out of the publication, use of, or reliance upon, this IEC Publication or any other IEC Publications.
8) Attention is drawn to the Normative references cited in this publication. Use of the referenced publications is indispensable for the correct application of this publication.
9) Attention is drawn to the possibility that some of the elements of this IEC Publication may be the subject of patent rights. IEC shall not be held responsible for identifying any or all such patent rights.

This redline version of the official IEC Standard allows the user to identify the changes made to the previous edition. A vertical bar appears in the margin wherever a change has been made. Additions are in green text, deletions are in strikethrough red text.

IEC 62541-10 has been prepared by subcommittee 65E: Devices and integration in enterprise systems, of IEC technical committee 65: Industrial-process measurement, control and automation.

This third edition cancels and replaces the second edition published in 2015.

This edition includes several clarifications and in addition the following significant technical changes with respect to the previous edition:

a) Changed ProgramType to ProgramStateMachineType. This is in line with the NodeSet (and thus implementations). In ProgramDiagnosticDataType: changed the definition of lastInputArguments and lastOutputArguments and added two additional fields for the argument values. Also changed StatusResult into StatusCode. Created new version of the type to ProgramDiagnostic2DataType.

b) Changed Optional modelling rule to OptionalPlaceHolder for Program control Methods. Following the clarification in IEC 62541-3, this now allows subtypes (or instances) to add arguments.

The text of this standard is based on the following documents:

FDIS	Report on voting
65E/719/FDIS	65E/735/RVD

Full information on the voting for the approval of this International Standard can be found in the report on voting indicated in the above table.

This document has been drafted in accordance with the ISO/IEC Directives, Part 2.

Throughout this document and the other parts of the IEC 62541 series, certain document conventions are used:

Italics are used to denote a defined term or definition that appears in Clause 3 in one of the parts of the series.

Italics are also used to denote the name of a service input or output parameter or the name of a structure or element of a structure that are usually defined in tables.

The *italicized terms and names* are also, with a few exceptions, written in camel-case (the practice of writing compound words or phrases in which the elements are joined without spaces, with each element's initial letter capitalized within the compound). For example the defined term is *AddressSpace* instead of *Address Space*. This makes it easier to understand that there is a single definition for *AddressSpace*, not separate definitions for Address and Space.

A list of all parts of the IEC 62541 series, published under the general title *OPC Unified Architecture*, can be found on the IEC website.

The committee has decided that the contents of this document will remain unchanged until the stability date indicated on the IEC website under "http://webstore.iec.ch" in the data related to the specific document. At this date, the document will be

- reconfirmed,
- withdrawn,
- replaced by a revised edition, or
- amended.

IMPORTANT – The 'colour inside' logo on the cover page of this publication indicates that it contains colours which are considered to be useful for the correct understanding of its contents. Users should therefore print this document using a colour printer.

OPC UNIFIED ARCHITECTURE –

Part 10: Programs

1 Scope

This part of IEC 62541 ~~is part of the overall OPC Unified Architecture (OPC UA) standard series and~~ defines the *information model* associated with *Programs* in the OPC Unified Architecture. This includes the description of the *NodeClasses,* standard *Properties*, *Methods* and *Events* and associated behaviour and information for *Programs*.

The complete Address Space model including all *NodeClasses* and *Attributes* is specified in IEC 62541-3. The *Services* such as those used to invoke the *Methods* used to manage *Programs* are specified in IEC 62541-4.

2 Normative references

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

IEC TR 62541-1, *OPC Unified Architecture – Part 1: Overview and Concepts*

IEC 62541-3~~:2015~~, *OPC Unified Architecture – Part 3: Address Space Model*

IEC 62541-4~~:2015~~, *OPC Unified Architecture – Part 4: Services*

IEC 62541-5~~:2015~~, *OPC Unified Architecture – Part 5: Information Model*

IEC 62541-7, *OPC Unified Architecture – Part 7: Profiles*

3 Terms, definitions and ~~conventions~~ abbreviated terms

3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in IEC TR 62541-1, IEC 62541-3 and the following apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

- IEC Electropedia: available at http://www.electropedia.org/
- ISO Online browsing platform: available at http://www.iso.org/obp

3.1.1
function
programmatic task performed by a *Server* or device, usually accomplished by computer code execution

3.1.2
finite state machine
sequence of states and valid state transitions along with the causes and effects of those state transitions that define the actions of a *Program* in terms of discrete stages

3.1.3
~~**ProgramType**~~
ProgramStateMachineType
type definition of a *Program* and is a subtype of the *FiniteStateMachineType*

3.1.4
program control method
Method having specific semantics designed for the control of a *Program* by causing a state transition

3.1.5
program invocation
unique *Object* instance of a *Program* existing on a *Server*

Note 1 to entry: A *Program Invocation* is distinguished from other *Object* instances of the same ~~*ProgramType*~~ *ProgramStateMachineType* by the object node's unique browse path.

3.2 Abbreviated terms

DA data access

FSM finite state machine

HMI human–machine interface

~~PCM Program Control Method~~

~~PGM Program~~

~~PI Program Invocation~~

UA Unified Architecture

4 Concepts

4.1 General

Integrated automation facilities manage their operations through the exchange of data and the coordinated invocation of system *Functions* as illustrated in Figure 1. *Services* are required to perform the data exchanges and to invoke the *Functions* that constitute system operation. These *Functions* may be invoked through Human Machine Interfaces, cell controllers, or other supervisory control and data acquisition type systems. OPC UA defines *Methods* and *Programs* as an interoperable way to advertise, discover, and request these *Functions*. They provide a normalizing mechanism for the semantic description, invocation, and result reporting of these *Functions*. Together *Methods* and *Programs* complement the other OPC UA *Services* and *ObjectTypes* to facilitate the operation of an automation environment using a client-server hierarchy.

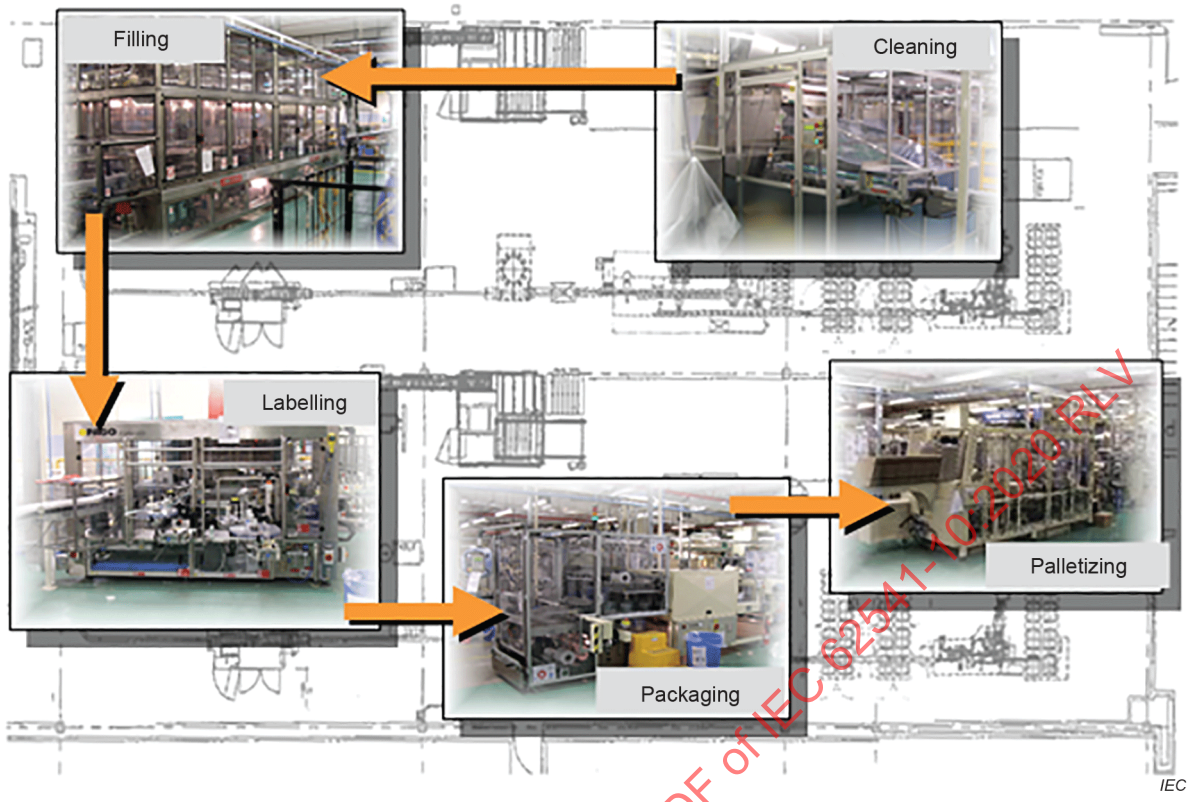


Figure 1 – Automation facility control

Methods and *Programs* model *Functions* typically have different scopes, behaviours, lifetimes, and complexities in *OPC Servers* and the underlying systems. These *Functions* are not normally characterized by the reading or writing of data which is accomplished with the OPC UA A*ttribute* service set.

Methods represent basic *Functions* in the *Server* that can be invoked by a *Client*. *Programs,* by contrast, model more complex and stateful functionality in the system. For example, a method call may be used to perform a calculation or reset a counter. A *Program* is used to run and control a batch process, execute a machine tool part program, or manage a domain download. *Methods* and their invocation mechanism are described in IEC 62541-3 and IEC 62541-4.

This document describes the extensions to, or specific use of, the core capabilities defined in IEC 62541-5 as required for *Programs*.

4.2 Programs

4.2.1 Overview

Programs are complex *Functions* in a *Server* or underlying system that can be invoked and managed by a *Client*. *Programs* can represent any level of functionality within a system or process in which *Client* control or intervention is required and progress monitoring is desired. Figure 2 illustrates the model.

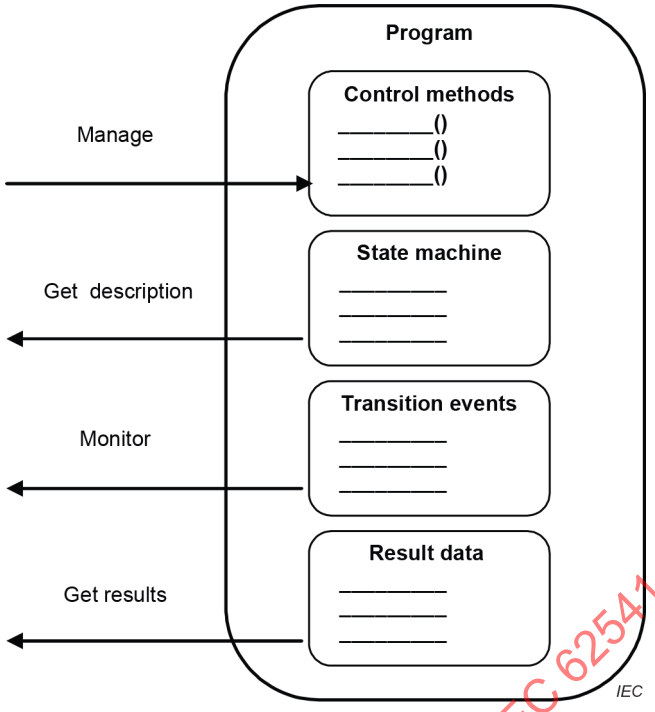


Figure 2 – Program illustration

Programs are stateful and transition through a prescribed sequence of states as they execute. Their behaviour is defined by a *Program Finite State Machine (PFSM)*. The elements of the PFSM describe the phases of a *Program's* execution in terms of valid transitions between a set of states, the stimuli or causes of those transitions, and the resultant effects of the transitions.

4.2.2 Security considerations

Since *Programs* can be used to perform advanced control algorithms or other actions, their use should be restricted to personnel with appropriate access rights. It is recommended that *AuditUpdateMethodEvents* are generated to allow monitoring the number of running *Programs* in addition to their execution frequency.

4.2.3 Program Finite State Machine

The states, transitions, causes and effects that compose the *Program Finite State Machine* are listed in Table 1 and illustrated in Figure 3.

Table 1 – Program Finite State Machine

No.	Transition name	Cause	From state	To state	Effect
1	HaltedToReady	Reset Method	Halted	Ready	Report Transition 1 Event/Result
2	ReadyToRunning	Start Method	Ready	Running	Report Transition 2 Event/Result
3	RunningToHalted	Halt Method or Internal (Error)	Running	Halted	Report Transition 3 Event/Result
4	RunningToReady	Internal	Running	Ready	Report Transition 4 Event/Result
5	RunningToSuspended	Suspend Method	Running	Suspended	Report Transition 5 Event/Result
6	SuspendedToRunning	Resume Method	Suspended	Running	Report Transition 6 Event/Result
7	SuspendedToHalted	Halt Method	Suspended	Halted	Report Transition 7 Event/Result
8	SuspendedToReady	Internal	Suspended	Ready	Report Transition 8 Event/Result
9	ReadyToHalted	Halt Method	Ready	Halted	Report Transition 9 Event/Result

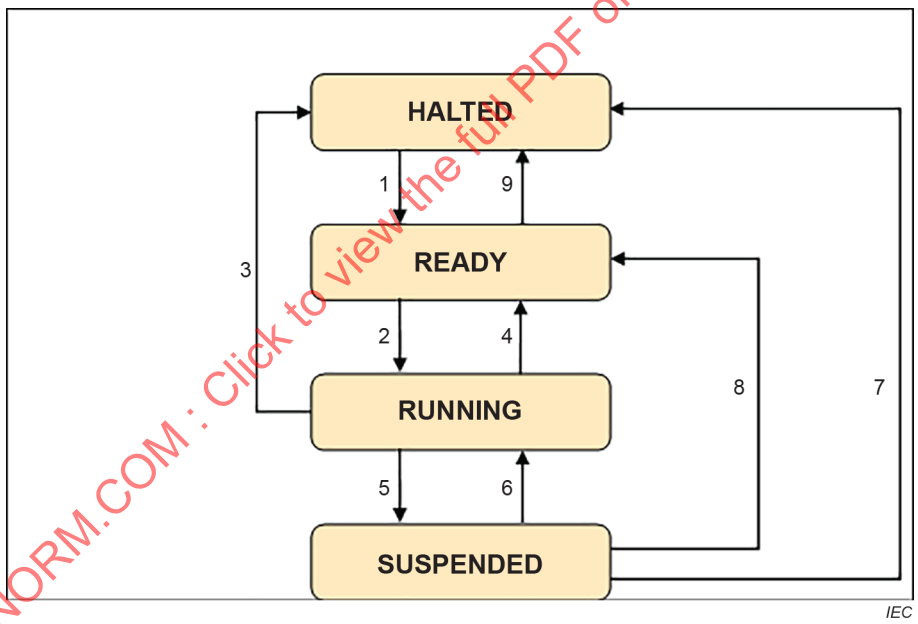


Figure 3 – Program states and transitions

4.2.4 Program states

A standard set of base states is defined for *Programs* as part of the *Program Finite State Machine.* These states represent the stages in which a *Program* can exist at an ~~instance~~ instant in time as viewed by a *Client*. This state is the *Program's* current state. All *Programs* shall support this base set. A *Program* may or may not require a *Client* action to cause the state to change. The states are formally defined in Table 2.

Table 2 – Program states

State	Description
Ready	The *Program* is properly initialized and may be started.
Running	The *Program* is executing making progress towards completion.
Suspended	The *Program* has been stopped prior to reaching a terminal state but may be resumed.
Halted	The *Program* is in a terminal or failed state, and it cannot be started or resumed without being reset.

The set of states defined to describe a *Program* can be expanded. *Program* sub states can be defined for the base states to provide more resolution of a process and to describe the cause and effect(s) of additional stimuli and transitions. Standards bodies and industry groups may extend the base *Program Finite State Model* to conform to various industry models. For example, the Halted state can include the sub states "Aborted" and "Completed" to indicate if the *Function* achieved a successful conclusion prior to the transition to Halted. Transitional states such as "Starting" or "Suspending" might also be extensions of the Running state, for example.

4.2.5 State transitions

A standard set of state transitions is defined for the *Program Finite State Machine*. These transitions define the valid changes to the *Program's* current state in terms of an initial state and a resultant state. The transitions are formally defined in Table 3.

Table 3 – Program state transitions

Transition no.	Transition name	Initial state	Resultant state
1	HaltedToReady	Halted	Ready
2	ReadyToRunning	Ready	Running
3	RunningToHalted	Running	Halted
4	RunningToReady	Running	Ready
5	RunningToSuspended	Running	Suspended
6	SuspendedToRunning	Suspended	Running
7	SuspendedToHalted	Suspended	Halted
8	SuspendedToReady	Suspended	Ready
9	ReadyToHalted	Ready	Halted

4.2.6 Program state transition stimuli

The stimuli or causes for a *Program's* state transitions can be internal to the *Server* or external. The completion of machining steps, the detection of an alarm condition, or the transmission of a data packet are examples of internal stimuli. *Methods* are an example of external stimuli. Standard *Methods* are defined which act as stimuli for the control of a *Program*.

4.2.7 Program Control Methods

Clients manage a *Program* by calling *Methods*. The *Methods* impact a *Program's* behaviour by causing specified state transitions. The state transitions dictate the actions performed by the *Program*. This document defines a set of standard *Program Control Methods*. These *Methods* provide sufficient means for a *Client* to run a *Program*.

Table 4 lists the set of defined *Program Control Methods*. Each *Method* causes transitions from specified states and shall be called when the *Program* is in one of those states.

Individual *Programs* can optionally support any subset of the *Program Control Methods*. For example, some *Programs* may not be permitted to suspend and so would not provide the *Suspend* and *Resume Methods*.

Programs can support additional user defined *Methods*. User defined *Methods* shall not change the behaviour of the base *Program Finite State Machine*.

Table 4 – Program Control Methods

Method Name	Description
Start	Causes the *Program* to transition from the Ready state to the Running state.
Suspend	Causes the *Program* to transition from the Running state to the Suspended state.
Resume	Causes the *Program* to transition from the Suspended state to the Running state.
Halt	Causes the *Program* to transition from the Ready, Running or Suspended state to the Halted state.
Reset	Causes the *Program* to transition from the Halted state to the Ready state.

~~*Program Control Methods* can include arguments that are used by the *Program*.~~ All *Program Control Methods* are defined with their *BrowseName* on the *ProgramStateMachineType* with the *OptionalPlaceholder ModellingRule*. As defined in IEC 62541-3, this rule allows the inclusion of *Arguments* to these Methods on subtypes or on instances. For example, a Start *Method* may include an options argument that specifies dynamic options used to determine some program behaviour. ~~The arguments can differ on each *ProgramType*.~~ The *Method Call* service specified in IEC 62541-4~~:2015, 5.11~~ defines a return status. This return status indicates the success of the *Program Control Method* or a reason for its failure.

4.2.8 Program state transition effects

A *Program's* state transition generally has a cause and also yields an effect. The effect is a by product of a *Program* state transition that can be used by a *Client* to monitor the progress of the *Program*. Effects can be internal or external. An external effect of a state transition is the generation of an *Event* notification. Each *Program* state transition is associated with a unique *Event*. These *Events* reflect the progression and trajectory of the *Program* through its set of defined states. The internal effects of a state transition can be the performance of some programmatic action such as the generation of data.

4.2.9 Program result data

4.2.9.1 Overview

Result data is generated by a running *Program*. The result data can be intermediate or final. Result data may be associated with specific *Program* state transitions.

4.2.9.2 Intermediate result data

Intermediate result data is transient and is generated by the *Program* in conjunction with non-terminal state transitions. The data items that compose the intermediate results are defined in association with specific *Program* state transitions. Their values are relevant only at the transition level.

Each *Program* state transition can be associated with different result data items. Alternately, a set of transitions can share a result data item. Percentage complete is an example of intermediate result data. The value of percentage complete is produced when the state transition occurs and is available to the *Client*.

Clients acquire intermediate result data by subscribing to *Program* state transition *Events*. The *Events* specify the data items for each transition. When the transition occurs, the generated *Event* conveys the result data values captured to the subscribed *Clients*. If no *Client* is monitoring the *Program*, intermediate result data may be discarded.

4.2.9.3 Terminal result data

Terminal result data is the final data generated by the *Program* as it ceases execution. Total execution time, number of widgets produced, and fault condition encountered are examples of terminal result data. When the *Program* enters the terminal state, this result data can be conveyed to the *Client* by the transition *Event*. Terminal result data is also available within the *Program* to be read by a *Client* after the program stops. This data persists until the *Program* Instance is rerun or deleted.

4.2.9.4 Monitoring Programs

Clients can monitor the activities associated with a *Program's* execution. These activities include the invocation of the management *Methods*, the generation of result data, and the progression of the *Program* through its states. *Audit Events* are provided for *Method Calls* and state transitions. These *Events* allow a record to be maintained of the *Clients* that interacted with any *Program* and the *Program* state transitions that resulted from that interaction.

4.2.10 Program lifetime

4.2.10.1 Overview

Programs can have different lifetimes. Some *Programs* may always be present on a *Server* while others are created and removed. Creation and removal can be controlled by a *Client* or may be restricted to local means.

A *Program* can be *Client* creatable. If a *Program* is *Client* creatable, then the *Client* can add the *Program* to the *Server*. The *Object Create Method* defined in IEC 62541-3~~:2015, 5.5.4,~~ is used to create the *Program* instance. The initial state of the *Program* can be Halted or Ready. Some *Programs*, for example, may require that a resource becomes available after its creation and before it is ready to run. In this case, it would be initialized in the Halted state and transition to Ready when the resource is delivered.

A *Program* can be *Client* removable. If the *Program* is *Client* removable, then the *Client* can delete the *Program* instance from the *Server*. The *DeleteNodes Service* defined in IEC 62541-4 is used to remove the *Program* Instance. The *Program* shall be in a Halted state to be removed. A *Program* may also be auto removable. An auto removable *Program* deletes itself when execution has terminated.

4.2.10.2 Program instances

Programs can be multiple instanced or single instanced. A *Server* can support multiple instances of a *Program* if these *Program* Instances can be run in parallel. For example, the *Program* may define a *Start Method* that has an input argument to specify which resource is acted upon by its *Functions*. Each instance of the *Program* is then started designating use of different resources. The *Client* can discover all instances of a *Program* that are running on a *Server*. Each instance of a *Program* is uniquely identified on the *Server* and is managed independently by the *Client*.

4.2.10.3 Program recycling

Programs can be run once or run multiple times (recycled). A *Program* that is run once will remain in the Halted state indefinitely once it has run. The normal course of action would be to delete it following the inspection of its terminal results.

Recyclable *Programs* may have a limited or unlimited cycle count. These *Programs* may require a reset step to transition from the Halted state to the Ready state. This allows for replenishing resources or reinitializing parameters prior to restarting the *Program*. The *Program Control Method* "*Reset*" triggers this state transition and any associated actions or effects.

5 Model

5.1 General

The *Program* model extends the *FiniteStateMachineType and* basic *ObjectType* models presented in IEC 62541-5. Each *Program* has a *Type Definition* that is the subtype of the *FiniteStateMachineType*. The ~~*ProgramType*~~ *ProgramStateMachineType* describes the *Finite State Machine* model supported by any *Program Invocation* of that type. The ~~*ProgramType*~~ *ProgramStateMachineType* also defines the property set that characterizes specific aspects of that *Program's* behaviour such as lifetime and recycling as well as specifying the result data that is produced by the *Program*.

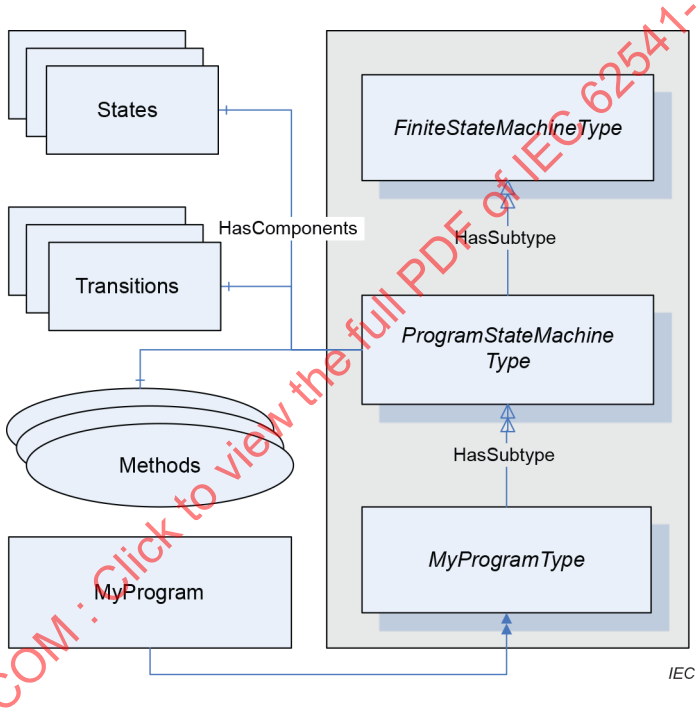


Figure 4 – Program Type

The base ~~*ProgramType*~~ *ProgramStateMachineType* defines the standard *Finite State Machine* specified for all *Programs*. This includes the states, transitions, and transition causes (*Methods*) and effects (*Events*). Subtypes of the base ~~*ProgramType*~~ *ProgramStateMachineType* can be defined to extend or more specifically characterize the behaviour of an individual *Program* as illustrated with "MyProgramType" in Figure 4.

5.2 ~~ProgramType~~ ProgramStateMachineType

5.2.1 Overview

The additional properties and components that compose the ~~*ProgramType*~~ *ProgramStateMachineType* are listed in Table 5. No ~~*ProgramType*~~ *ProgramStateMachineType* specific semantics are assigned to the other base *ObjectType* or *FiniteStateMachineType Attributes* or *Properties*.

Table 5 – ~~ProgramType~~ ProgramStateMachineType

Attribute	Value				
	Includes all attributes specified for the FiniteStateMachineType				
BrowseName	~~ProgramType~~ ProgramStateMachineType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**Data Type**	**TypeDefinition**	**Modelling Rule**
HasProperty	Variable	Creatable	Boolean	PropertyType	–
HasProperty	Variable	Deletabe	Boolean	PropertyType	Mandatory
HasProperty	Variable	AutoDelete	Boolean	PropertyType	Mandatory
HasProperty	Variable	RecycleCount	Int32	PropertyType	Mandatory
HasProperty	Variable	InstanceCount	UInt32	PropertyType	–
HasProperty	Variable	MaxInstanceCount	UInt32	PropertyType	–
HasProperty	Variable	MaxRecycleCount	UInt32	PropertyType	–
HasComponent	Variable	ProgramDiagnostic	ProgramDiagnostic2DataType	ProgramDiagnostic2Type	Optional
HasComponent	Object	Halted		StateType	~~Mandatory~~ –
HasComponent	Object	Ready		StateType	~~Mandatory~~ –
HasComponent	Object	Running		StateType	~~Mandatory~~ –
HasComponent	Object	Suspended		StateType	~~Mandatory~~ –
HasComponent	Object	HaltedToReady		TransitionType	~~Mandatory~~ –
HasComponent	Object	ReadyToRunning		TransitionType	~~Mandatory~~ –
HasComponent	Object	RunningToHalted		TransitionType	~~Mandatory~~ –
HasComponent	Object	RunningToReady		TransitionType	~~Mandatory~~ –
HasComponent	Object	RunningToSuspended		TransitionType	~~Mandatory~~ –
HasComponent	Object	SuspendedToRunning		TransitionType	~~Mandatory~~ –
HasComponent	Object	SuspendedToHalted		TransitionType	~~Mandatory~~ –
HasComponent	Object	SuspendedToReady		TransitionType	~~Mandatory~~ –
HasComponent	Object	ReadyToHalted		TransitionType	~~Mandatory~~ –
HasComponent	Method	Start			OptionalPlaceholder
HasComponent	Method	Suspend			OptionalPlaceholder
HasComponent	Method	Reset			OptionalPlaceholder
HasComponent	Method	Halt			OptionalPlaceholder
HasComponent	Method	Resume			OptionalPlaceholder
HasComponent	Object	FinalResultData		BaseObjectType	Optional

5.2.2 ~~ProgramType~~ ProgramStateMachineType Properties

The *Creatable Property* is a boolean that specifies if *Program Invocations* of this ~~*ProgramType*~~ *ProgramStateMachineType* can be created by a *Client*. If False, these *Program Invocations* are persistent or may only be created by the *Server*.

The *Deletable Property* is a boolean that specifies if a *Program Invocation* of this ~~*ProgramType*~~ *ProgramStateMachineType* can be deleted by a *Client*. If False, these *Program Invocations* can only be deleted by the *Server*.

The *AutoDelete Property* is a boolean that specifies if *Program Invocations* of this ~~*ProgramType*~~ *ProgramStateMachineType* are removed by the *Server* when execution terminates. If False, these *Program Invocations* persist on the *Server* until they are deleted by the *Client*. When the *Program Invocation* is deleted, any result data associated with the instance is also removed.

The *RecycleCount Property* is an unsigned integer that specifies the number of times a *Program Invocation* of this type has been recycled or restarted from its starting point (not resumed). Note that the Reset Method may be required to prepare a *Program* to be restarted.

The *MaxRecycleCount Property* is an integer that specifies the maximum number of times a *Program Invocation* of this type can be recycled or restarted from its starting point (not resumed). If the value is less than 0, then there is no limit to the number of restarts. If the value is zero, then the Program may not be recycled or restarted.

The *InstanceCount Property* is an unsigned integer that specifies the number of *Program Invocations* of this type that currently exist.

The *MaxInstanceCount Property* is an integer that specifies the maximum number of *Program Invocations* of this type that can exist simultaneously on this *Server*. If the value is less than 0, then there is no limit.

5.2.3 ~~ProgramType~~ ProgramStateMachineType components

5.2.3.1 Overview

The ~~*ProgramType*~~ *ProgramStateMachineType* components consist of a set of *References* to the *Object* instances of *StateTypes*, *TransitionTypes*, *EventTypes* and the *Methods* that collectively define the *Program FiniteStateMachine.*

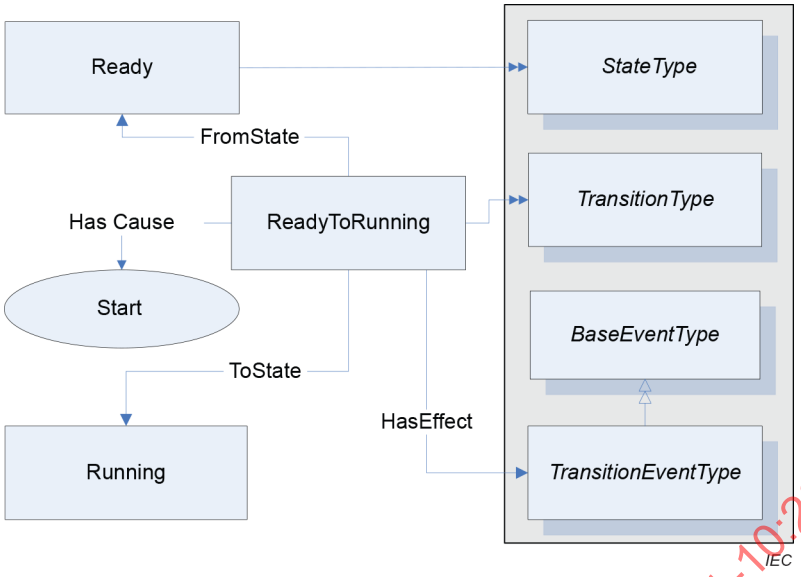


Figure 5 – Program FSM References

Figure 5 illustrates the component *References* that define the associations between two of the ~~*ProgramType's*~~ *ProgramStateMachineType's* states, Ready and Running. The complementary ReferenceTypes have been omitted to simplify the illustration.

5.2.3.2 ~~ProgramType~~ ProgramStateMachineType states

Table 6 specifies the ~~*ProgramType's*~~ *ProgramStateMachineType's* state *Objects*. These *Objects* are instances of the *StateType* defined in IEC 62541-5~~:2015, Annex B~~. Each state is assigned a unique *StateNumber* value. Subtypes of the ~~*ProgramType*~~ *ProgramStateMachineType* can add references from any state to a subordinate or nested *StateMachine Object* to extend the *FiniteStateMachine*.

Table 6 – Program states

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	NOTES
States					
Halted	HasProperty	StateNumber	~~1~~ 11	PropertyType	
	ToTransition	HaltedToReady		TransitionType	
	FromTransition	RunningToHalted		TransitionType	
	FromTransition	SuspendedToHalted		TransitionType	
	FromTransition	ReadyToHalted		TransitionType	
Ready	HasProperty	StateNumber	~~2~~ 12	PropertyType	
	FromTransition	HaltedToReady		TransitionType	
	ToTransition	ReadyToRunning		TransitionType	
	FromTransition	RunningToReady		TransitionType	
	ToTransition	ReadyToHalted		TransitionType	
Running	HasProperty	StateNumber	~~3~~ 13	PropertyType	
	ToTransition	RunningToHalted		TransitionType	
	ToTransition	RunningToReady		TransitionType	
	ToTransition	RunningToSuspended		TransitionType	
	FromTransition	ReadyToRunning		TransitionType	
	FromTransition	SuspendedToRunning		TransitionType	
Suspended	HasProperty	StateNumber	~~4~~ 14	PropertyType	
	ToTransition	SuspendedToRunning		TransitionType	
	ToTransition	SuspendedToHalted		TransitionType	
	ToTransition	SuspendedToReady		TransitionType	
	FromTransition	RunningToSuspended		TransitionType	

The Halted state is the idle state for a *Program*. It can be an initial state or a terminal state. As an initial state, the *Program Invocation* cannot begin execution due to conditions at the *Server*. As a terminal state, Halted can indicate either a failed or completed *Program*. A subordinate state or result can be used to distinguish the nature of the termination. The Halted state references four *Transition Objects*, which identify the allowed state transitions to the Ready state and from the Ready, Running, and Suspended states.

The Ready state indicates that the *Program* is prepared to begin execution. *Programs* that are ready to begin upon their creation may transition immediately to the Ready state. The Ready state references four *Transition Objects,* which identify the allowed state transitions to the Running and Halted states and from the Halted and Ready states.

The Running state indicates that the *Program* is actively performing its *Function*. The Running state references five *Transition Objects,* which identify the allowed state transitions to the Halted, Ready, and Suspended states and from the Ready and Suspended states.

The Suspended state indicates that the *Program* has stopped performing its *Function*, but retains the ability to resume the *Function* at the point at which it was executing when suspended. The Suspended state references four *Transition Objects*, which identify the allowed state transitions to the Ready, Running, and Halted state and from the Ready state.

5.2.3.3 ~~ProgramType~~ ProgramStateMachineType transitions

~~*ProgramType*~~ *ProgramStateMachineType* Transitions are instances of the *TransitionType* defined in IEC 62541-5~~:2015, Clause B.4~~ which also includes the definitions of the ToState, FromState, HasCause, and HasEffect references used. Table 7 specifies the transitions defined for the ~~*ProgramType*~~ *ProgramStateMachineType*. Each transition is assigned a unique *TransitionNumber*. The Notes column indicates when a cause is referencing *Methods* and when effects are optional.

Table 7 – Program transitions

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	Notes
Transitions					
HaltedToReady	HasProperty	TransitionNumber	1	PropertyType	
	ToState	Ready		StateType	
	FromState	Halted		StateType	
	HasCause	Reset			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
ReadyToRunning	HasProperty	TransitionNumber	2	PropertyType	
	ToState	Running		StateType	
	FromState	Ready		StateType	
	HasCause	Start			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
RunningToHalted	HasProperty	TransitionNumber	3	PropertyType	
	ToState	Halted		StateType	
	FromState	Running		StateType	
	HasCause	Halt			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
RunningToReady	HasProperty	TransitionNumber	4	PropertyType	
	ToState	Ready		StateType	
	FromState	Runnning		StateType	
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	Notes
RunningToSuspended	HasProperty	TransitionNumber	5	PropertyType	
	ToState	Running		StateType	
	FromState	Suspended		StateType	
	HasCause	Suspend			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
SuspendedToRunning	HasProperty	TransitionNumber	6	PropertyType	
	ToState	Running		StateType	
	FromState	Suspended		StateType	
	HasCause	Resume			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
SuspendedToHalted	HasProperty	TransitionNumber	7	PropertyType	
	ToState	Halted		StateType	
	FromState	Suspended		StateType	
	HasCause	Halt			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
SuspendedToReady	HasProperty	TransitionNumber	8	PropertyType	
	ToState	Ready		StateType	
	FromState	Suspended		StateType	
	HasCause	Reset			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
ReadyToHalted	HasProperty	TransitionNumber	9	PropertyType	
	ToState	Halted		StateType	
	FromState	Ready		StateType	
	HasCause	Halt			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			

The *HaltedToReady* transition specifies the transition from the Halted to Ready states. It may be caused by the *Reset Method.*

The *ReadyToRunning* transition specifies the transition from the Ready to Running states. It is caused by the *Start Method.*

The *RunningToHalted* transition specifies the transition from the Running to Halted states. It is caused by the *Halt Method.*

The *RunningToReady* transition specifies the transition from the Running to Ready states.

The *RunningToSuspended* transition specifies the Transition from the Running to Suspended states. It is caused by the *Suspend Method.*

The *SuspendedToRunning* transition specifies the transition from the Suspended to Running states. It is caused by the *Resume Method.*

The *SuspendedToHalted* transition specifies the transition from the Suspended to Halted states. It is caused by the *Halt Method.*

The *SuspendedToReady* transition specifies the transition from the Suspended to Ready states. It is caused internally.

The *ReadyToHalted* transition specifies the transition from the Ready to Halted states. It is caused by the *Halt Method.*

Two *HasEffect* references are specified for each *Program* transition. These effects are *Events* of *ProgramTransitionEventType* and *AuditProgramTransitionEventType* defined in 5.2.5. The *ProgramTransitionEventType* notifies *Clients* of the *Program* transition and conveys result data. The *AuditProgramTransitionEventType* is used to audit transitions that result from *Program Control Methods*. The "*Audit Server Facet*" *Profile* defined in IEC 62541-7 requires support of the *AuditProgramTransitionEventType*.

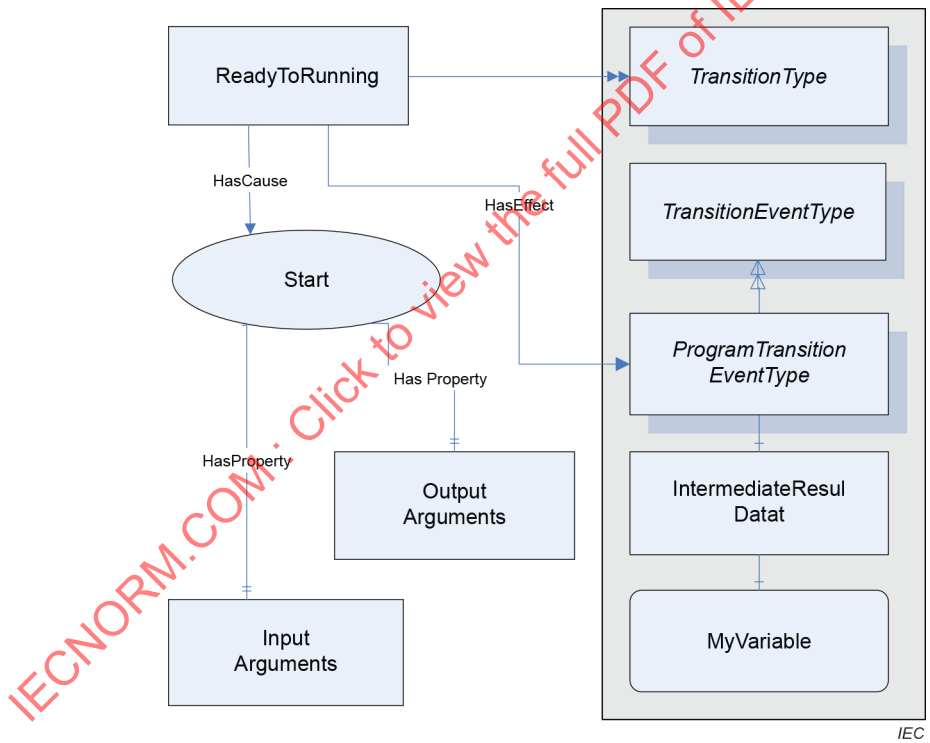


Figure 6 – ~~ProgramType~~ ProgramStateMachineType causes and effects

5.2.4 ~~ProgramType~~ ProgramStateMachineType causes (Methods)

5.2.4.1 Overview

The ~~*ProgramType*~~ *ProgramStateMachineType* includes references to the Causes of specific *Program* state transitions. These causes refer to *Method* instances. *Programs* that do not support *a Program Control Method* will omit the Causes reference to that *Method* from the ~~*ProgramType*~~ *ProgramStateMachineType* references. If a *Method's Causes* reference is omitted from the ~~*ProgramType*~~ *ProgramStateMachineType* then a *Client* cannot cause the associated state transition. The *Method* instances referenced by the ~~*ProgramType*~~

ProgramStateMachineType identify the *InputArguments* and *OutputArguments* required for the *Method* calls to *Program Invocations* of that ~~*ProgramType*~~ *ProgramStateMachineType*. Table 8 specifies the *Methods* defined as Causes for ~~*ProgramTypes*~~ *ProgramStateMachineTypes*. Figure 6 illustrates the *References* associating the components and *Properties* of *Methods* and *Events* with Program transitions.

Table 8 – ~~ProgramType~~ ProgramStateMachineType causes

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	NOTES
Causes					
Start	HasProperty	InputArguments		PropertyType	Optional
	HasProperty	OutputArguments		PropertyType	Optional
Suspend	HasProperty	InputArguments		PropertyType	Optional
	HasProperty	OutputArguments		PropertyType	Optional
Resume	HasProperty	InputArguments		PropertyType	Optional
	HasProperty	OutputArguments		PropertyType	Optional
Halt	HasProperty	InputArguments		PropertyType	Optional
	HasProperty	OutputArguments		PropertyType	Optional
Reset	HasProperty	InputArguments		PropertyType	Optional
	HasProperty	OutputArguments		PropertyType	Optional

The *Start Method* causes the *ReadyToRunning Program* transition.

The *Suspend Method* causes the *RunningToSuspended Program* transition.

The *Resume Method* causes the *SuspendedToRunning Program* transition.

The *Halt Method* causes the *RunningToHalted, SuspendedToHalted, or ReadyToHalted Program* transition depending on the current state of the *Program*.

The *Reset Method* causes the *HaltedToReady Program* transition.

5.2.4.2 Standard attributes

The *Executable Method* attribute indicates if a method can currently be executed. For *Program Control Methods*, this means that the owning *Program* has a current state that supports the transition caused by the *Method.*

5.2.4.3 Standard properties

~~5.2.4.3.1 Overview~~

Methods can reference a set of *InputArguments*. For each ~~*ProgramType*~~ *ProgramStateMachineType*, a set of *InputArguments* may be defined for the supported *Program Control Methods*. The data passed in the arguments supplements the information required by the *Program* to perform its *Function*. All calls to a *Program Control Method* for

each *Program Invocation* of that ~~*ProgramType*~~ *ProgramStateMachineType* shall pass the specified arguments.

Methods can reference a set of *OutputArguments*. For each ~~*ProgramType*~~ *ProgramStateMachineType*, a set of *OutputArguments* is defined for the supported *Program Control Methods*. All calls to a *Program Control Method* for each *Program Invocation* of that ~~*ProgramType*~~ *ProgramStateMachineType* shall pass the specified arguments.

5.2.5 ~~ProgramType~~ ProgramStateMachineType effects (Events)

5.2.5.1 Overview

The ~~*ProgramType*~~ *ProgramStateMachineType* includes component references to the Effects of each of the *Program's* state transitions. These Effects are *Events*. Each *Transition* shall have a *HasEffect Reference* to a *ProgramTransitionEventType* and can have an *AuditProgramTransitionEventType*. When the transition occurs, *Event* notifications of the referenced type are generated for subscribed *Clients*. The *Program Invocation* may serve as the *EventNotifier* for these *Events* or an owning *Object* or the *Server Object* may provide the notifications.

ProgramTransitionEventTypes provide the means for delivering result data and confirming state transitions for subscribed *Clients* on each defined *Program State Transition*. The *AuditProgramTransitionEventType* allows the auditing of changes to the *Program*'s state in conjunction with *Client Method Calls*.

5.2.5.2 ProgramTransitionEventType

The *ProgramTransitionEventType* is a subtype of the *TransitionEventType*. It is used with *Programs* to acquire intermediate or final results or other data associated with a state transition. A *Program* can have a unique *ProgramTransitionEventType* definition for any transition. Each *ProgramTransitionEventType* specifies the *IntermediateResult* data specific to the designated state transition on that ~~*ProgramType*~~ *ProgramStateMachineType*. Each transition can yield different intermediate result data. Table 9 specifies the *ProgramTransitionEventType*.

Table 10 identifies the *ProgramTransitionEventTypes* that are specified for ~~*ProgramTypes*~~ *ProgramStateMachineTypes*.

Table 9 – ProgramTransitionEventType

Attribute	Value				
BrowseName	ProgramTransitionEventType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the base *TransitionEventType* defined in IEC 62541-5~~:2015, B.4.16~~.					
HasComponent	Object	IntermediateResult		BaseObjectType	Optional

TransitionNumber identifies the *Program* transition that triggered the *Event*.

FromStateNumber identifies the state before the *Program* transition.

ToStateNumber identifies the state after the *Program* transition.

The *IntermediateResult* is an *Object* that aggregates a set of *Variables* whose values are relevant for the *Program* at the instant of the associated transition. The *ObjectType* for the

IntermediateResult specifies the collection of *Variables* using a set of *HasComponent References*.

Table 10 – ProgramTransitionEvents

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	Notes
Effects					
HaltedToReadyEvent					
	HasProperty	TransitionNumber	1	PropertyType	
	HasProperty	FromStateNumber	1	PropertyType	
	HasProperty	ToStateNumber	2	PropertyType	
	HasComponent	IntermediateResults		ObjectType	Optional
ReadyToRunningEvent					
	HasProperty	TransitionNumber	2	PropertyType	
	HasProperty	FromStateNumber	2	PropertyType	
	HasProperty	ToStateNumber	3	PropertyType	
	HasComponent	IntermediateResults		ObjectType	Optional
RunningToHaltedEvent					
	HasProperty	TransitionNumber	3	PropertyType	
	HasProperty	FromStateNumber	3	PropertyType	
	HasProperty	ToStateNumber	1	PropertyType	
	HasComponent	IntermediateResults		ObjectType	Optional
RunningToReadyEvent					
	HasProperty	TransitionNumber	4	PropertyType	
	HasProperty	FromStateNumber	3	PropertyType	
	HasProperty	ToStateNumber	2	PropertyType	
	HasComponent	IntermediateResults		ObjectType	Optional
RunningToSuspendedEvent					
	HasProperty	TransitionNumber	5	PropertyType	
	HasProperty	FromStateNumber	3	PropertyType	
	HasProperty	ToStateNumber	4	PropertyType	
	HasComponent	IntermediateResults		ObjectType	Optional
SuspendedToRunningEvent					
	HasProperty	TransitionNumber	6	PropertyType	
	HasProperty	FromStateNumber	4	PropertyType	
	HasProperty	ToStateNumber	3	PropertyType	
	HasComponent	IntermediateResults		ObjectType	Optional
SuspendedToHaltedEvent					
	HasProperty	TransitionNumber	7	PropertyType	
	HasProperty	FromStateNumber	4	PropertyType	

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	Notes
	HasProperty	ToStateNumber	1	PropertyType	
	HasComponent	IntermediateResults		ObjectType	Optional
SuspendedToReadyEvent					
	HasProperty	TransitionNumber	8	PropertyType	
	HasProperty	FromStateNumber	4	PropertyType	
	HasProperty	ToStateNumber	2	PropertyType	
	HasComponent	IntermediateResults		ObjectType	Optional
ReadyToHaltedEvent					
	HasProperty	TransitionNumber	9	PropertyType	
	HasProperty	FromStateNumber	2	PropertyType	
	HasProperty	ToStateNumber	1	PropertyType	
	HasComponent	IntermediateResults		ObjectType	Optional

5.2.6 AuditProgramTransitionEventType

The *AuditProgramTransitionEventType* is a subtype of the *AuditUpdateStateEventType*. It is used with *Programs* to provide a means to audit the *Program State* transitions associated with any *Client* invoked *Program Control Method*. *Servers* shall generate *AuditProgramTransitionEvents* if they support the *Audit Server Facet Profile* defined in IEC 62541-7.

Table 11 specifies the definition of the *AuditProgramTransitionEventType*.

Table 11 – AuditProgramTransitionEventType

Attribute	Value				
BrowseName	AuditProgramTransitionEventType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *AuditUpdateStateEventType* defined in IEC 62541-5~~:, B.4.17~~.					
HasProperty	Variable	TransitionNumber	UInt32	PropertyType	Mandatory

This *EventType* inherits all *Properties* of the *AuditUpdateStateEventType* defined in IEC 62541-5~~:2015, B.4.17~~, except as noted below.

The *Status Property*, specified in IEC 62541-5~~:2015, 6.4.3~~, identifies whether the state transition resulted from a Program Control Method call (set *Status* to TRUE) or not (set *Status* to FALSE).

The *SourceName*, specified in IEC 62541-5~~:2015, 6.4.2~~, identifies the *Method* causing the *Program* transition when it is the result of a *Client* invoked *ProgramControlMethod*. The *SourceName* is prefixed with "Method/" and the name of the *ProgramControlMethod*, "Method/Start" for example.

The *ClientUserId Property*, specified in IEC 62541-5~~:2015, 6.4.3~~, identifies the user of the *Client* that issued the *Program Control Method* if it is associated with this *Program* state transition.

The *ActionTimeStamp Property*, specified in IEC 62541-5~~:2015, 6.4.3~~ "AuditEventType", identifies when the time the *Program* state transition that resulted in the *Event* being generated occurred.

The *TransitionNumber Property* is a *Variable* that identifies the transition that triggered the *Event*.

5.2.7 FinalResultData

The *FinalResultData ObjectType* specifies the *VariableTypes* that are preserved when the *Program* has completed its *Function*. The *ObjectType* includes *a HasComponent* for a *VariableType* of each *Variable* that comprises the final result data.

5.2.8 ProgramDiagnostic2 DataType

This structure contains elements that chronicle the *Program Invocation's* activity and can be used to aid in the diagnosis of *Program* problems.

NOTE The original ProgramDiagnosticDataType had flaws. To avoid collisions with existing implementations, a new version with name ProgramDiagnostic2DataType has been created.

Its composition is defined in Table 12.

Table 12 – ProgramDiagnostic2DataType structure

Name	Type	Description
ProgramDiagnostic2DataType	structure	
createSessionId	NodeId	The *CreateSessionId* contains the *SessionId* of the *Session*on which the call to the *Create Method* was issued to create the *Program Invocation*.
createClientName	String	The *CreateClientName* is the name of the *Client*of the *Session*that created the *Program Invocation*.
invocationCreationTime	UtcTime	The *InvocationCreationTime* identifies the time the *Program Invocation* was created.
lastTransitionTime	UtcTime	The *LastTransitionTime* identifies the time of the last *Program* state transition that occurred.
lastMethodCall	String	The *LastMethodCall* identifies the last *Program Method* called on the *Program Invocation*.
lastMethodSessionId	NodeId	The *LastMethodSessionId* contains the *SessionId* of the *Session*on which the last *Program Control Method* call to the *Program Invocation* was issued.
lastMethodInputArguments	Argument[]	The *LastMethodInputArguments* ~~preserves the values of~~ provides the input arguments on the last *Program Method* call.
lastMethodOutputArguments	Argument[]	The *LastMethodOutputArguments* ~~preserves~~ provides ~~the values of~~ the output arguments on the last *Program Method* call.
lastMethodInputValues	BaseDataType[]	The *LastMethodInputValues* preserves the values of the input arguments on the last *Program Method* call. The size and order of this list matches the size and order of the *lastMethodInputArguments* field.
lastMethodOutputValues	BaseDataType[]	The *LastMethodOutputValues* preserves the values of the output arguments on the last *Program Method* call. The size and order of this list matches the size and order of the *lastMethodOutputArguments* field.
lastMethodCallTime	UtcTime	The *LastMethodCallTime* identifies the time of the last Method call to the *Program Invocation*.
lastMethodReturnStatus	~~StatusResult~~ StatusCode	The *LastMethodReturnStatus* preserves the value of the return status for the last *Program Control Method* requested for *this Program Invocation.*

Its representation in the *AddressSpace* is defined in Table 13.

Table 13 – ProgramDiagnostic2DataType definition

Attributes	Value
BrowseName	ProgramDiagnostic2DataType

5.2.9 ProgramDiagnostic2Type VariableType

This *VariableType* aggregates simple *Variables* using simple *DataTypes* that reflect the elements of the ProgramDiagnosticDataType structure. Its *DataVariables* have the same semantic as defined in in 5.2.8.

NOTE The original ProgramDiagnosticType VariableType had the same flaws as the structure. To avoid collisions with existing implementations, a new version with name ProgramDiagnostic2Type has been created.

The *VariableType* is formally defined in Table 14.

Table 14 – ProgramDiagnostic2Type VariableType

Attribute	Value			
BrowseName	ProgramDiagnostic2Type			
IsAbstract	False			
References	**NodeClass**	**BrowseName**	**DataType / TypeDefinition**	**Modelling Rule**
Subtype of the BaseDataVariableType defined in IEC 62541-5.				
HasComponent	Variable	CreateSessionId	NodeId	Mandatory
HasComponent	Variable	CreateClientName	String	Mandatory
HasComponent	Variable	InvocationCreationTime	UtcTime	Mandatory
HasComponent	Variable	LastTransitionTime	UtcTime	Mandatory
HasComponent	Variable	LastMethodCall	String	Mandatory
HasComponent	Variable	LastMethodSessionId	NodeId	Mandatory
HasComponent	Variable	LastMethodInputArguments	Argument[]	Mandatory
HasComponent	Variable	LastMethodOutputArguments	Argument[]	Mandatory
HasComponent	Variable	LastMethodInputValues	BaseDataType[]	Mandatory
HasComponent	Variable	LastMethodOutputValues	BaseDataType[]	Mandatory
HasComponent	Variable	LastMethodCallTime	UtcTime	Mandatory
HasComponent	Variable	LastMethodReturnStatus	~~StatusResult~~ StatusCode	Mandatory

Annex A
(informative)

Program example

A.1 Overview

This example illustrates the use of a *Program* to manage a domain download into a control system as depicted in Figure A.1. The download requires the segmented transfer of control operation data from a secondary storage device to the local memory within a control system.

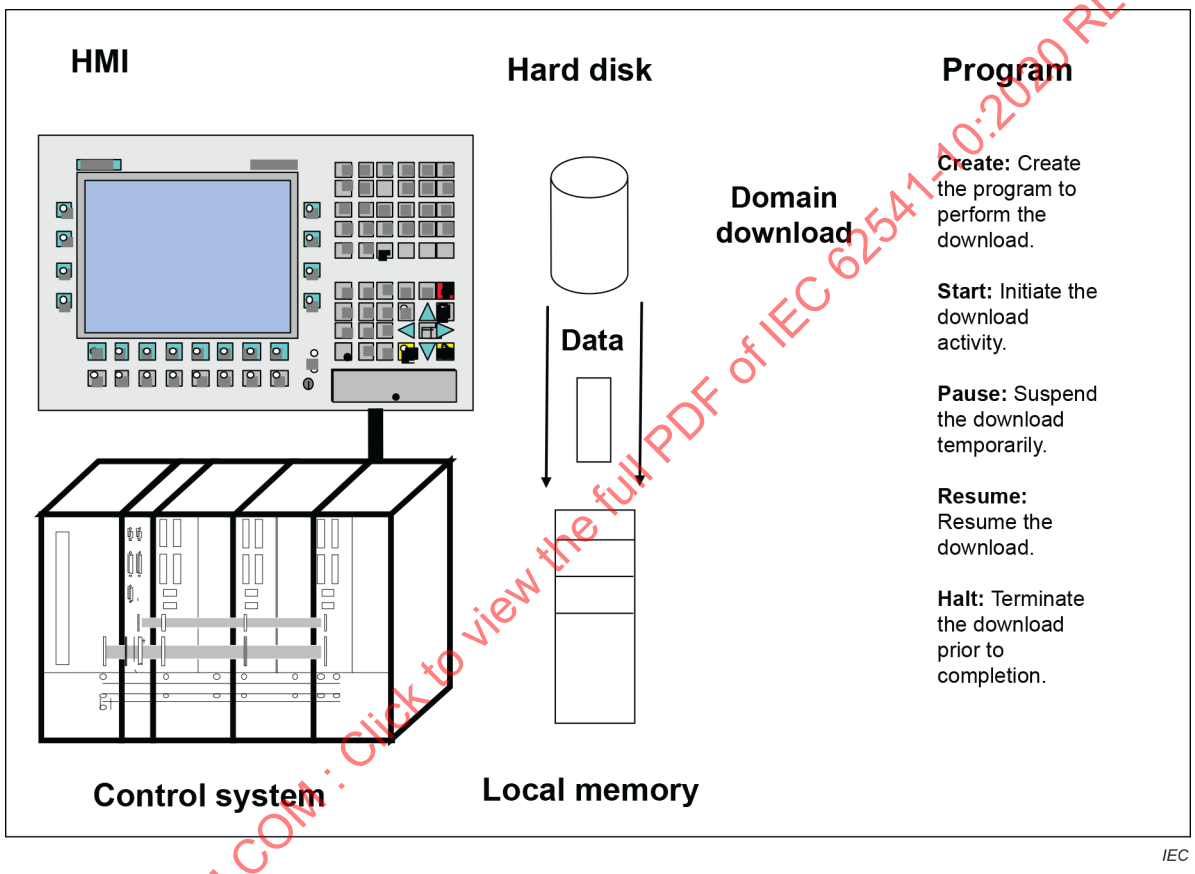


Figure A.1 – Program example

The domain download has a source and a target location which are identified when the download is initiated. Each time a segment of the domain is successfully transferred the *Client* is notified and informed of the amount of data that has been downloaded. The *Client* is also notified when the download is finished. The percentage of the total data received is reported periodically while the download continues. If the download fails, the cause of the failure is reported. At the completion of the download, the performance information is kept at the *Server*.

A.2 DomainDownload Program

A.2.1 General

The *Client* uses the "DomainDownload" *Program* to manage and monitor the download of a domain at the *Server*.

A.2.2 DomainDownload states

The basic state model for the DomainDownload *Program* is presented in Figure A.2. The *Program* has three primary states, Ready, Running, and Halted which are aligned with the standard states of a ~~*ProgramType*~~ *ProgramStateMachineType*. Additionally, the *DomainDownloadType* extends the ~~*ProgramType*~~ *ProgramStateMachineType* by defining subordinate *State Machines* for the *Program's* Running and Halted states. The subordinate states describe the download operations in greater detail and allow the *Client* to monitor the activity of the download at a finer resolution.

An instance (*Program Invocation*) of a DomainDownload *Program* is created by the *Client* each time a download is to be performed. The instance exists until explicitly removed by the *Client*. The initial state of the *Program* is Ready and the terminal state is Halted. The DomainDownload can be temporarily suspended and then resumed or aborted. Once halted, the program may not be restarted.

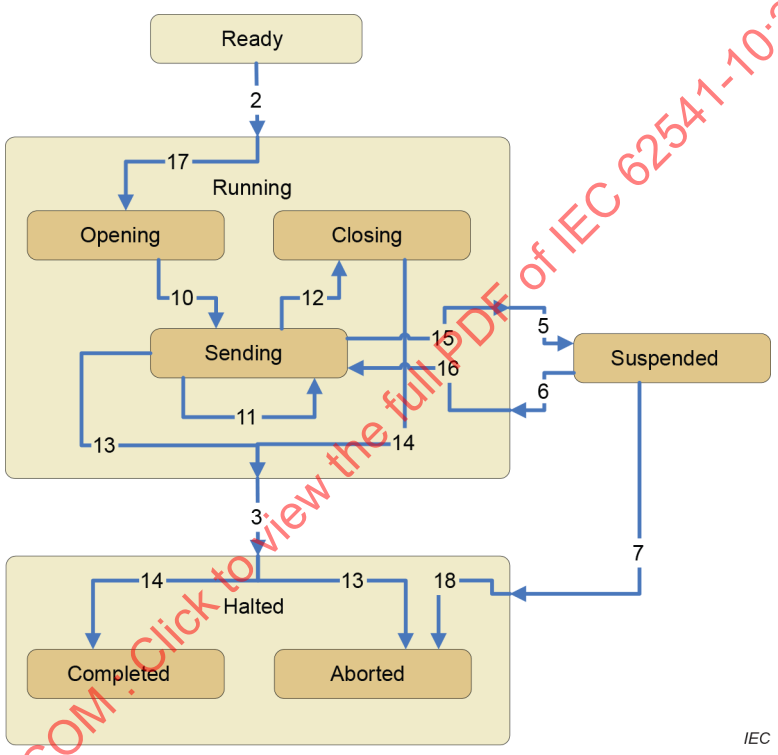


Figure A.2 – DomainDownload state diagram

The sequence of state transitions is illustrated in Figure A.2. Once the download is started, The *Program* progresses to the Opening state. After the source of the data is opened, a sequence of transfers occurs in the Sending state. When the transfer completes the *Objects* are closed in the Closing state. If the transfer is terminated before all of the data is downloaded or an error is encountered then the download is halted and the *Program* transitions to the Aborted state; otherwise the *Program* halts in the Completed state. The states are presented in Table A.1 along with the state transitions.

A.2.3 DomainDownload transitions

The valid state transitions specified for the DomainDownload *Program* are specified in Table A.1. Each of the transitions defines a start state and end state for the transition and is identified by a unique number. Five of the transitions are from the base ~~*ProgramType*~~ *ProgramStateMachineType* and retain the transition identifier numbers specified for *Programs*. The additional transitions relate the base *Program* states with the subordinate states defined for the DomainDownload. These states have been assigned unique transition identifier numbers that distinguish them from the base *Program* transition identifiers. In cases

where transitions occur between substates and the *Program's* base states, two transitions are specified. One transition identifies the base state change and a second substate change. For example, ReadytoRunning and ToOpening occur at the same time.

Table A.1 also specifies the defined states, causes for the transitions, and the effects of each transition. *Program Control Methods* are used by the *Client* to "run" the DomainDownload. The *Methods* cause or trigger the specified transitions. The transition effects are the specified *EventTypes* which notify the *Client* of *Program* activity.

Table A.1 – DomainDownload states

No.	Transition name	Cause	From State	To State	Effect
2	ReadyToRunning	Start Method	Ready	Running	Report Transition 2 Event/Result
3	RunningToHalted	Halt Method/Error or Internal.	Running	Halted	Report Transition 3 Event/Result
5	RunningToSuspended	Suspend Method	Running	Suspended	Report Transition 5 Event/Result
6	SuspendedToRunning	Resume Method	Suspended	Running	Report Transition 6 Event/Result
7	SuspendedToHalted	Halt Method	Suspended	Halted	Report Transition 7 Event/Result
10	OpeningToSending	Internal	Opening	Sending	Report Transition 10 Event/Result
11	SendingToSending	Internal	Sending	Sending	Report Transition 11 Event/Result
12	SendingToClosing	Internal	Sending	Closing	Report Transition 12 Event/Result
13	SendingToAborted	Halt Method/Error	Opening	Aborted	Report Transition 13 Event/Result
14	ClosingToCompleted	Internal	Closing	Completed	Report Transition 14 Event/Result
15	SendingToSuspended	Suspend Method	Sending	Suspended	Report Transition 16 Event/Result
16	SuspendedToSending	Resume Method	Suspended	Sending	Report Transition 17 Event/Result
18	SuspendedToAborted	Halt Method	Suspended	Aborted	Report Transition 18 Event/Result
17	ToOpening	Internal	Ready	Opening	Report Transition 19 Event/Result

A.2.4 DomainDownload Methods

A.2.4.1 General

Four standard *Program Methods* are specified for running the DomainDownload *Program, Start, Suspend, Resume, and Halt.* No additional *Methods* are specified. The base behaviours of these *Methods* are defined by the ~~*ProgramType*~~ *ProgramStateMachineType*. The *Start Method* initiates the download activity and passes the source and destination locations for the transfer. The *Suspend Method* is used to pause the activity temporarily. The *Resume Method* reinitiates the download, when paused. The *Halt Method* aborts the download. Each of the *Methods* causes a *Program* state transition and a substate transition. The specific state transition depends on the current state at the time the *Method* is called. If a *Method Call* is made when the DomainDownload is in a state for which that *Method* has no associated transition, the *Method* returns an error status indicating invalid state for the *Method*.

A.2.4.2 Method Arguments

The *Start Method* specifies three input arguments to be passed when it is called: Domain Name, DomainSource, and DomainDestination. The other *Methods* require no input arguments. No output arguments are specified for the DomainDownload *Methods*. The resultant error status for the *Program* is part of the *Call Service*.

A.2.5 DomainDownload Events

A.2.5.1 General

A *ProgramTransitionEventType* is specified for each of the DomainDownload *Program* transitions. The *EventTypes* trigger a specific *Event* notification to the *Client* when the associated state transition occurs in the running *Program* Instance. The *Event* notification identifies the transition. The SendingToSending state transition also includes intermediate result data.

A.2.5.2 Event information

The SendingToSending *Program* transition *Event* relays intermediate result data to the *Client* along with the notification. Each time the transition occurs, data items describing the amount and percentage of data transferred are sent to the *Client*.

A.2.5.3 Final result data

The DomainDownload *Program* retains final result data following a completed or aborted download. The data includes the total transaction time and the size of the domain. In the event of an aborted download, the reason for the termination is retained.

A.2.6 DomainDownload model

A.2.6.1 Overview

The OPC UA model for the DomainDownload *Program* is presented in ~~the following tables and figures~~ A.2.6.2. Collectively they define the components that constitute this *Program*. For clarity, the figures present a progression of portions of the model that complement the contents of the tables and illustrate the *Program's* composition.

The type definition for the DomainDownload *Program* precisely represents the behaviour of the *Program* in terms of OPC UA components. These components can be browsed by a *Client* to interpret or validate the actions of the *Program*.

A.2.6.2 DomainDownloadType

The DomainDownloadType is a subtype derived from the ~~*ProgramType*~~ *ProgramStateMachineType*. It specifies the use or non-use of optional ~~*ProgramType*~~ *ProgramStateMachineType* components, valid extensions such as subordinate *State Machines*, and constrained attribute values applied to instances of DomainDownload *Programs*.

Table A.2 specifies the optional and extended components defined by the DomainDownload Type. Note the references to two sub *State Machine Types*, *TransferStateMachine* and *FinishStateMachine*. The DomainDownloadType omits references to the *Reset Program Control Method* and its associated state transition (HaltedToReady), which it does not support.

Table A.2 – DomainDownload Type

Attribute	Value				
	Includes all non-optional attributes specified for the ~~ProgramType~~ *ProgramStateMachineType*				
BrowseName	DomainDownloadType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**Data Type**	**TypeDefinition**	**Modelling Rule**
HasComponent	Object	TransferStateMachine		StateMachineType	Mandatory
HasComponent	Object	FinishStateMachine		StateMachineType	Mandatory
HasComponent	Variable	ProgramDiagnostic		ProgramDiagnostic2Type	Mandatory
HasComponent	Object	ReadyToRunning		TransitionType	~~Mandatory~~ –
HasComponent	Object	RunningToHalted		TransitionType	~~Mandatory~~ –
HasComponent	Object	RunningToSuspended		TransitionType	~~Mandatory~~ –
HasComponent	Object	SuspendedToRunning		TransitionType	~~Mandatory~~ –
HasComponent	Object	SuspendedToHalted		TransitionType	~~Mandatory~~ –
HasComponent	Method	Start			Mandatory
HasComponent	Method	Suspend			Mandatory
HasComponent	Method	Halt			Mandatory
HasComponent	Method	Resume			Mandatory
HasComponent	Object	FinalResultData		BaseObjectType	Mandatory

Table A.3 specifies the *Transfer State Machine type* that is a sub *State Machine* of the DomainDownload *Program Type*. This definition identifies the *StateTypes* that compose the substates for the *Program's* Running *StateType*.

Table A.3 – Transfer State Machine Type

Attribute	Value				
	Includes all attributes specified for the FiniteStateMachineType				
BrowseName	TransferStateMachineType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**Data Type**	**TypeDefinition**	**Modelling Rule**
HasComponent	Object	Opening		StateType	~~Mandatory~~ –
HasComponent	Object	Sending		StateType	~~Mandatory~~ –
HasComponent	Object	Closing		StateType	~~Mandatory~~ –
HasComponent	Object	ReadyToOpening		TransitionType	~~Mandatory~~ –
HasComponent	Object	OpeningToSending		TransitionType	~~Mandatory~~ –
HasComponent	Object	SendingToClosing		TransitionType	~~Mandatory~~ –
HasComponent	Object	SendingToAborted		TransitionType	~~Mandatory~~ –
HasComponent	Object	SendingToSuspended		TransitionType	~~Mandatory~~ –
HasComponent	Object	SuspendedToSending		TransitionType	~~Mandatory~~ –
HasComponent	Method	Start			Mandatory
HasComponent	Method	Suspend			Mandatory
HasComponent	Method	Halt			Mandatory
HasComponent	Method	Resume			Mandatory

Table A.3 specifies the *StateTypes* associated with the Transfer State Machine Type. All of these states are substates of the *Running* state of the base ~~*ProgramType*~~ *ProgramStateMachineType*.

The Opening state is the preparation state for the domain download.

The Sending state is the activity state for the transfer in which the data is moved from the source to destination.

The Closing state is the cleanup phase of the download.

Table A.4 defines the states of the *TransferStateMachineType*.

Table A.4 – Transfer State Machine – states

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	NOTES
States					
Opening	HasProperty	StateNumber	5	PropertyType	
	ToTransition	OpeningToSending		TransitionType	
	FromTransition	ToOpening		TransitionType	
	ToTransition	OpeningToSending		TransitionType	
Sending	HasProperty	StateNumber	6	PropertyType	
	FromTransition	OpeningToSending		TransitionType	
	ToTransition	SendingToSending		TransitionType	
	ToTransition	SendingToClosing		TransitionType	
	ToTransition	SendingToSuspended		TransitionType	
	FromTransition	ToSending		TransitionType	
Closing	HasProperty	StateNumber	7	PropertyType	
	ToTransition	ClosingToCompleted		TransitionType	
	ToTransition	ClosingToAborted		TransitionType	
	FromTransition	SendingToClosing		TransitionType	

Table A.5 specifies the *Finish State Machine Type* that is a substate machine of the DomainDownload ~~*ProgramType*~~ *ProgramStateMachineType*. This definition identifies the *StateTypes* that compose the substates for the *Program's* Halted *StateType*.

Table A.5 – Finish State Machine Type

Attribute	Value				
	Includes all attributes specified for the FiniteStateMachineType				
BrowseName	TransferStateMachineType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**Data Type**	**TypeDefinition**	**Modelling Rule**
HasComponent	Object	Completed		StateType	~~Mandatory~~ –
HasComponent	Object	Aborted		StateType	~~Mandatory~~ –

Table A.6 specifies the *StateTypes* associated with the *Finish State Machine Type*. Note these are final states and that they have no associated transitions between them.

Table A.6 – Finish State Machine – states

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	NOTES
States					
Aborted	HasProperty	StateNumber	8	PropertyType	
	FromTransition	OpeningToAborted		TransitionType	
	FromTransition	ClosingToAborted		TransitionType	
Completed	HasProperty	StateNumber	9	PropertyType	
	FromTransition	ClosingToCompleted		TransitionType	

The Aborted state is the terminal state that indicates an incomplete or failed domain download operation.

The Completed state is the terminal state that indicates a successful domain download.

Table A.7 specifies the constraining behaviour of a DomainDownload.

Table A.7 – DomainDownload Type Property Attributes variable values

NodeClass	BrowseName	Data Type	Data Value	Modelling Rule
Variable	Creatable	Boolean	True	–
Variable	Deletable	Boolean	True	Mandatory
Variable	AutoDelete	Boolean	False	Mandatory
Variable	RecycleCount	Int32	0	Mandatory
Variable	InstanceCount	UInt32	PropertyType	–
Variable	MaxInstanceCount	UInt32	500	–
Variable	MaxRecycleCount	UInt32	0	–

A DomainDownload *Program Invocation* can be created and also destroyed by a *Client*. The *Program Invocation* will not delete itself when halted, but will persist until explicitly removed by the *Client*. A DomainDownload *Program Invocation* cannot be reset to restart. The *Server* will support up to 500 concurrent DomainDownload *Program Invocations*.

Figure A.3 presents a partial DomainDownloadType model that illustrates the association between the states and the DomainDownload, Transfer, and Finish State Machines. Note that the current state number for the substate machines is only valid when the DomainDownload active base state references the substate machine, Running for the Transfer current state and Halted for the Finish current state.

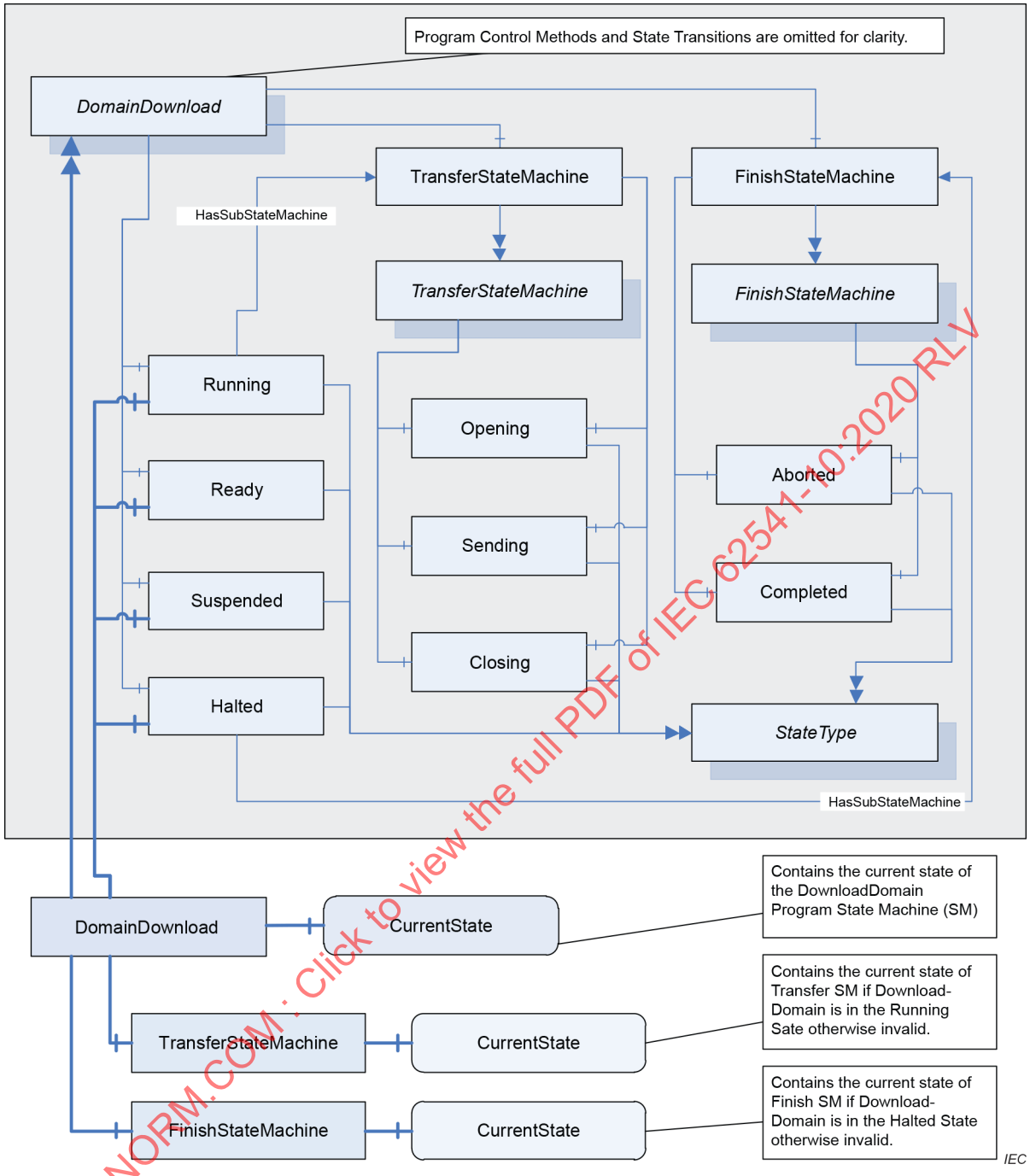


Figure A.3 – DomainDownloadType partial state model

Table A.8 specifies the *ProgramTransitionTypes* that are defined in addition to the *ProgramTransitionTypes* specified for *Programs* in Table 7. These types associate the Transfer and Finish substate machine states with the states of the base *Program*.

Table A.8 – Additional DomainDownload transition types

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	NOTES
Transitions					
ToSending	HasProperty	TransitionNumber	10	PropertyType	
	ToState	Sending		StateType	
	FromState	Opening		StateType	
	HasCause	Start			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
SendingToSending	HasProperty	TransitionNumber	11	PropertyType	
	ToState	Sending		StateType	
	FromState	Sending		StateType	
	HasEffect	ProgramTransitionEventType			
SendingToClosing	HasProperty	TransitionNumber	12	PropertyType	
	ToState	Closing		StateType	
	FromState	Sending		StateType	
	HasEffect	ProgramTransitionEventType			
SendingToAborted	HasProperty	TransitionNumber	13	PropertyType	
	ToState	Aborted		StateType	
	FromState	Sending		StateType	
	HasCause	Halt			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
ClosingToCompleted	HasProperty	TransitionNumber	14	PropertyType	
	ToState	Completed		StateType	
	FromState	Closing		StateType	
	HasEffect	ProgramTransitionEventType			
SendingToSuspended	HasProperty	TransitionNumber	15	PropertyType	
	ToState	Suspended		StateType	
	FromState	Sending		StateType	
	HasCause	Suspend			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	NOTES
Transitions					
SuspendedToSending	HasProperty	TransitionNumber	16	PropertyType	
	ToState	Sending		StateType	
	FromState	Suspended		StateType	
	HasCause	Resume			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
SuspendedToAborted	HasProperty	TransitionNumber	18	PropertyType	
	ToState	Aborted		StateType	
	FromState	Suspended		StateType	
	HasCause	Halt			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
ReadyToOpening	HasProperty	TransitionNumber	17	PropertyType	
	ToState	Opening		StateType	
	FromState	Ready		StateType	
	HasCause	Start			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			

Figure A.4 through Figure A.10 illustrate portions of the DomainDownloadType model. In each figure, the referenced tates, *Methods*, transitions, and *EventTypes* are identified for one or two state transitions.

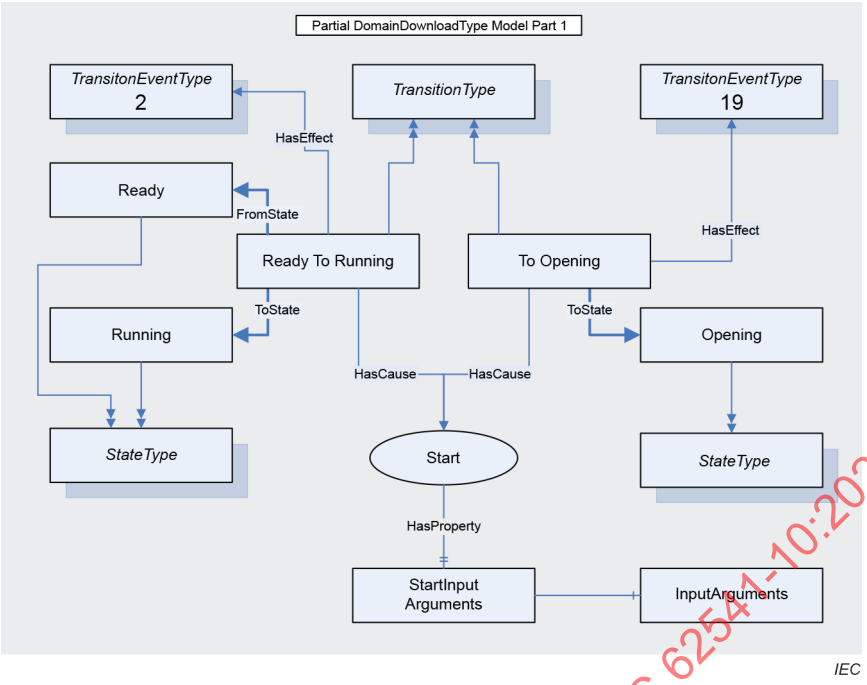


Figure A.4 – Ready To Running model

Figure A.4 illustrates the model for the ReadyToRunning *Program* transition. The transition is caused by the *Start Method*. The *Start Method* requires three input arguments. The *Method Call* service is used by the *Client* to invoke the *Start Method* and pass the arguments. When successful, the *Program Invocation* enters the Running state and the subordinate Transfer Opening state. The *Server* issues two *Event* notifications, ReadyToRunning (2), and ToOpening (19).

Table A.9 – Start Method additions

Attribute	**Value**				
BrowseName	Start				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArgument	Argument[]	PropertyType	–

Table A.9 specifies that the *Start Method* for the *DomainDownloadType* requires input arguments. Table A.10 identifies the *Start Arguments* required.

Table A.10 – StartArguments

Name	Type	Value
Argument 1	structure	
name	String	SourcePath
dataType	NodeId	StringNodeId
valueRank	Int32	-1 (-1 = scalar)
arrayDimensions	UInt32[]	null
description	LocalizedText	The source specifier for the domain
Argument 2	structure	
Name	String	DestinationPath
dataType	NodeId	StringNodeId
valueRank	Int32	-1 (-1 = scalar)
arrayDimensions	UInt32[]	null
description	LocalizedText	The destination specifier for the domain
Argument 3	structure	
name	String	DomainName
dataType	NodeId	StringNodeId
arrayDimensions	UInt32[]	null
valueRank	Int32	-1 (-1 = scalar)
description	LocalizedText	The name of the domain

Figure A.5 illustrates the model for the Opening To Sending and the Sending to Closing *Program* transitions. As specified in the transition table, these state transitions require no *Methods* to occur, but rather are driven by the internal actions of the *Server*. *Events* are generated for each state transition (10 to 12), when they occur.

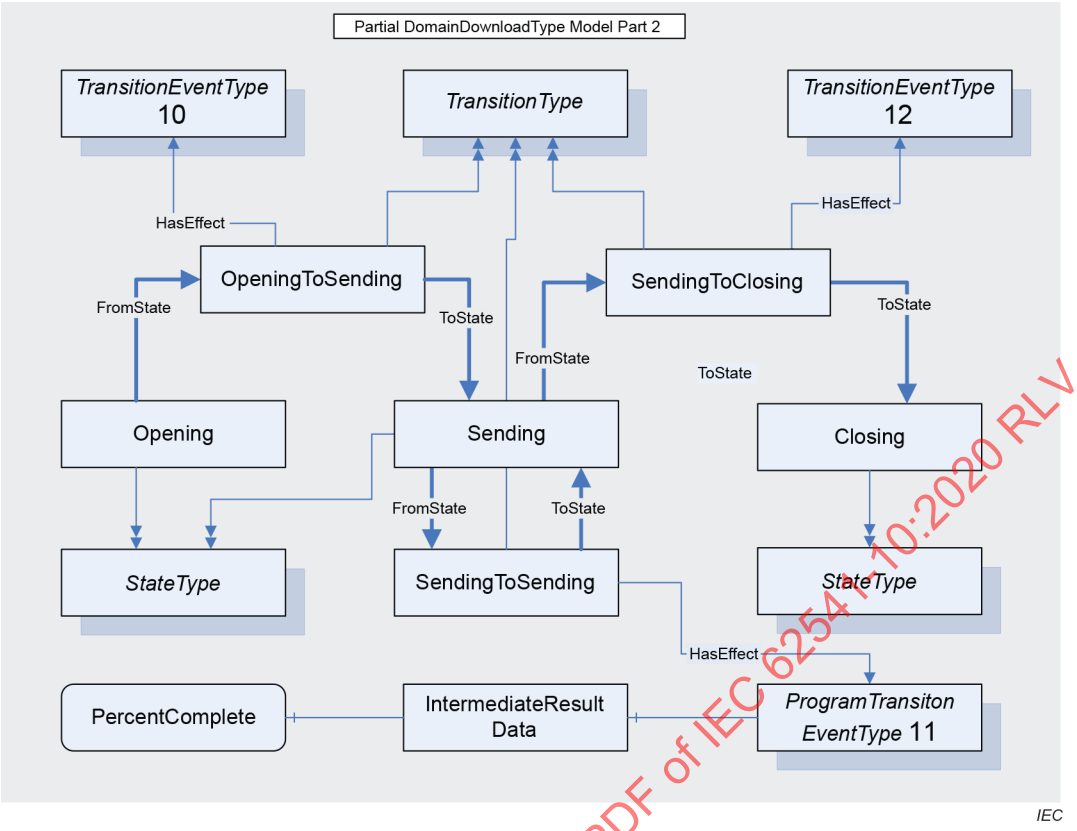


Figure A.5 – Opening To Sending To Closing model

Notice that a state transition can initiate and terminate at the same state (Sending). In this case the transition serves a purpose. The *ProgramTransitionEventType* effect referenced by the SendingToSending state transition has an *IntermediateResultData Object Reference*. The *IntermediateResultData Object* serves to identify two *Variables* whose values are obtained each time the state transition occurs. The values are sent to the *Client* with the *Event* notification. Table A.11 defines the *IntermediateResults ObjectType* and Table A.12 defines the *Variables* of the *ObjectType*.

Table A.11 – IntermediateResults Object

Attribute	Value				
	Includes all attributes specified for the ObjectType				
BrowseName	IntermediateResults				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**Data Type**	**TypeDefinition**	**Modelling Rule**
HasComponent	Variable	AmountTransferred	Long	VariableType	Mandatory
HasComponent	Variable	PercentageTransferred	Long	VariableType	Mandatory

Table A.12 – Intermediate result data Variables

Intermediate Result Variables	Type	Value
Variable 1	Structure	
Name	String	AmountTransferred
dataType	NodeId	StringNodeId
description	LocalizedText	Bytes of domain data transferred.
Variable 2	Structure	
Name	String	PercentageTransferred
dataType	NodeId	StringNodeId
description	LocalizedText	Percentage of domain data transferred.

The model for the Running To Suspended state transition is illustrated in Figure A.6. The cause for this transition is the *Suspend Method.* The *Client* can pause the download of domain data to the control. The transition from Running to Suspended invokes the *Event* generation for *TransitionEventTypes* 5 and 16. Note that there is no longer a valid current state for the Transfer State Machine.

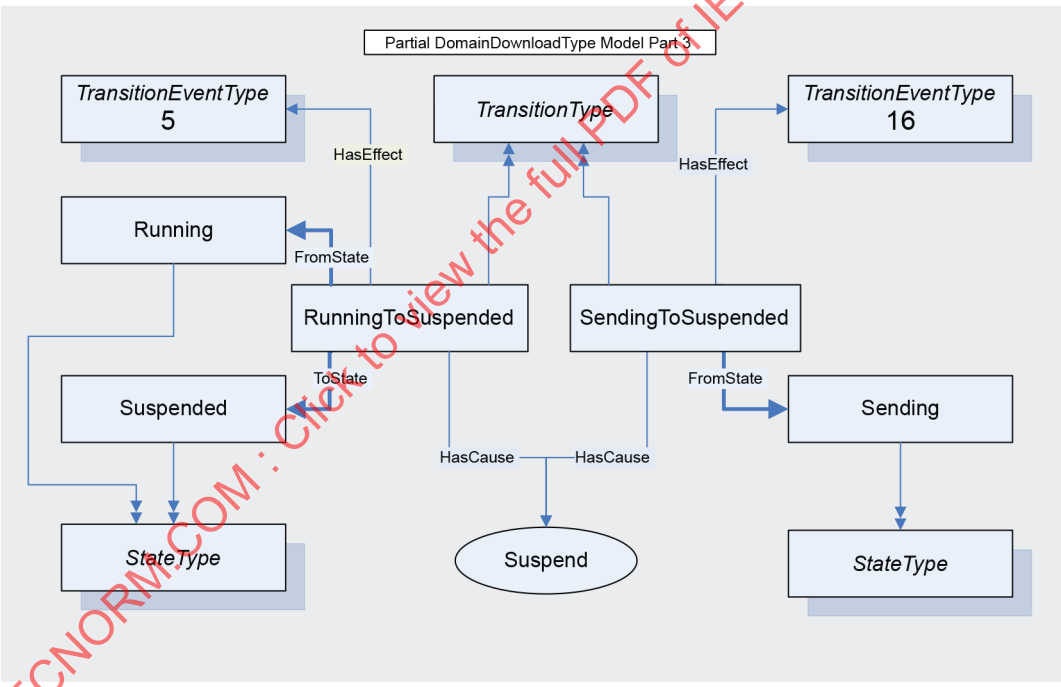


Figure A.6 – Running To Suspended model

The model for the SuspendedToRunning state transition is illustrated in Figure A.7. The cause for this transition is the *Resume Method.* The *Client* can resume the download of domain data to the control. The transition from Suspended to Running generates the *Event* for *TransitionEventTypes* 6 and 17. Now that the Running state is active, the Sending state of the Transfer State Machine is again specified for the *CurrentStateNumber*.

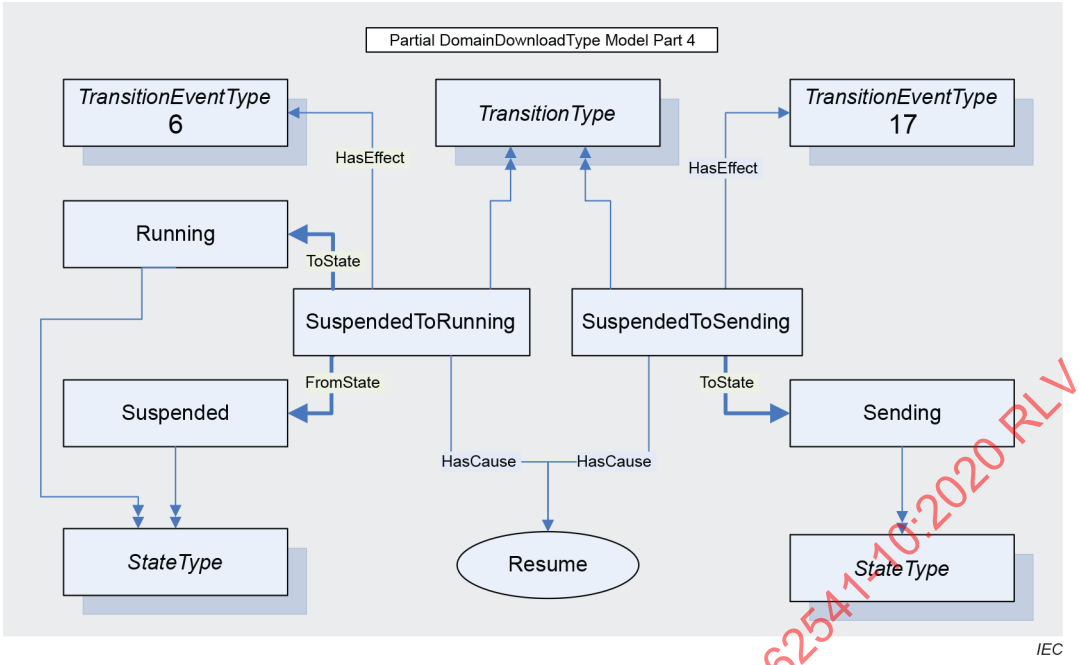


Figure A.7 – Suspended To Running model

The model for the Running To Halted state transition for an abnormal termination of the domain download is illustrated in Figure A.8. The cause for this transition is the *Halt Method.* The *Client* can terminate the download of domain data to the control. The transition from Running To Halted generates the *Event* for *TransitionEventTypes* 3 and 15. The *TransitionEventType* 15 indicates the transition from the Sending state as the Running State ends and then to the Aborted state as the Halted state is entered.

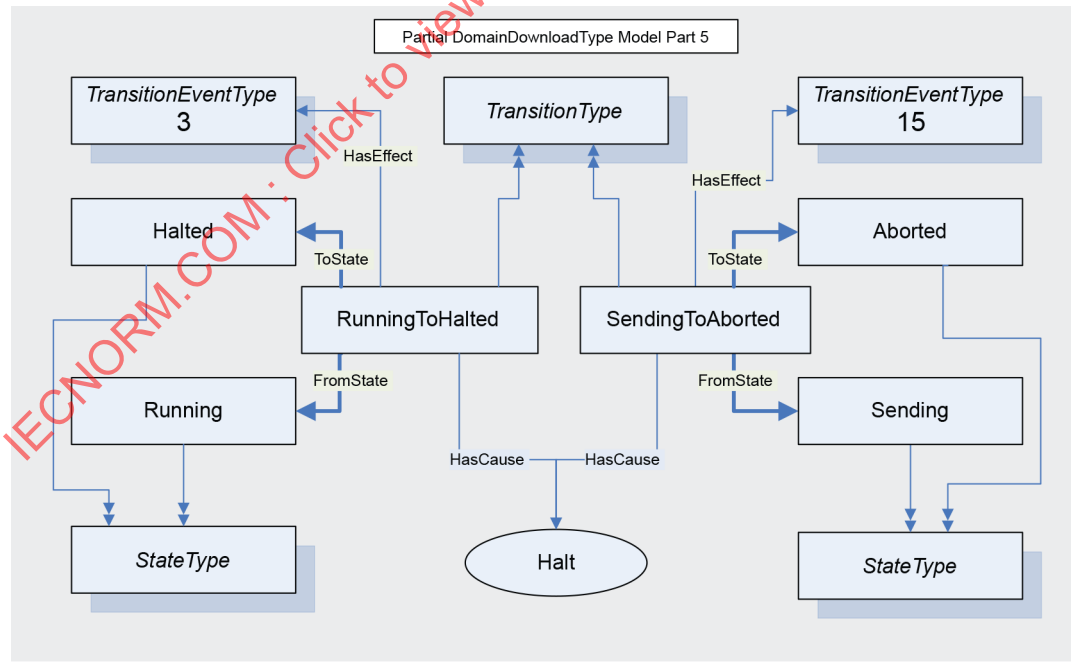


Figure A.8 – Running To Halted – Aborted model

Figure A.9 illustrates the model for the Suspended To Halted state transition for an abnormal termination of the domain download. The cause for this transition is the *Halt Method.* The *Client* can terminate the download of domain data to the control while it is suspended. The

transition from SuspendedToHalted invokes the *Event* notifiers for *TransitionEventTypes* 7 and 18.

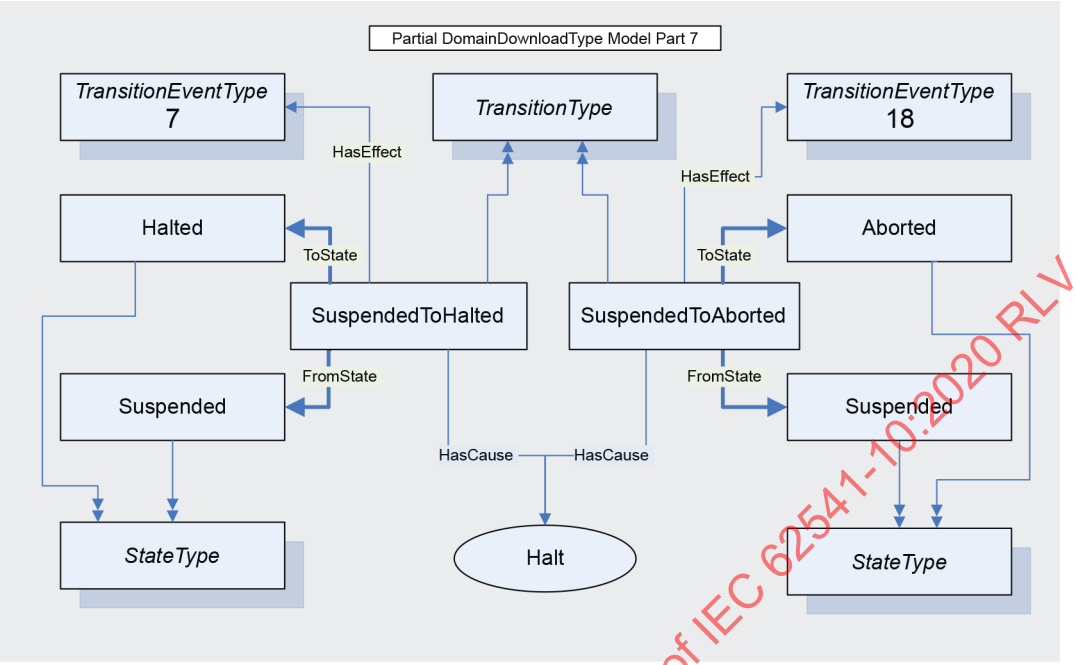


Figure A.9 – Suspended To Aborted model

The model for the Running To Completed state transition for a normal termination of the domain download is illustrated in Figure A.10. The cause for this transition is internal. The transition from Closing To Halted generates the Event for *TransitionEventTypes* 3 and 14. The *TransitionEventType* 14 indicates the transition from the Closing state as the Running state ends and then to the Completed state as the Halted state is entered.

The DomainDownloadType includes a component reference to a *FInalResultData Object*. This *Object* references *Variables* that persists information about the domain download once it has completed. This data can be read by *Clients* who are not subscribed to *Event* notifications. The result data is described in Table A.13 ~~and the *Variables* in Table A.14~~.

Table A.13 – FinalResultData

Attribute	Value				
	Includes all attributes specified for the ObjectType				
BrowseName	FinalResultData				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**Data Type**	**TypeDefinition**	**Modelling Rule**
HasComponent	Variable	DownloadPerformance	~~Long~~ Double	VariableType	Mandatory
HasComponent	Variable	FailureDetails	~~Long~~ String	VariableType	Mandatory

The Domain Download net transfer data rate and detailed reason for aborted downloads are retained as final result data for each *Program Invocation*.

~~Table A.14 – Final result Variables~~

~~Final Result Variables~~	~~Type~~	~~Value~~
~~Variable 1~~	~~Structure~~	
~~Name~~	~~String~~	~~DownloadPerformance~~
~~dataType~~	~~NodeId~~	~~Double~~
~~description~~	~~LocalizedText~~	~~Data rate for domain data transferred~~
~~Variable 2~~	~~Structure~~	
~~Name~~	~~String~~	~~FailureDetails~~
~~dataType~~	~~NodeId~~	~~StringNodeId~~
~~description~~	~~LocalizedText~~	~~Description of reason for abort~~

DownloadPerformance provides the data rate in seconds for domain data transferred.

FailureDetails provides a descriptive reason for an abort.

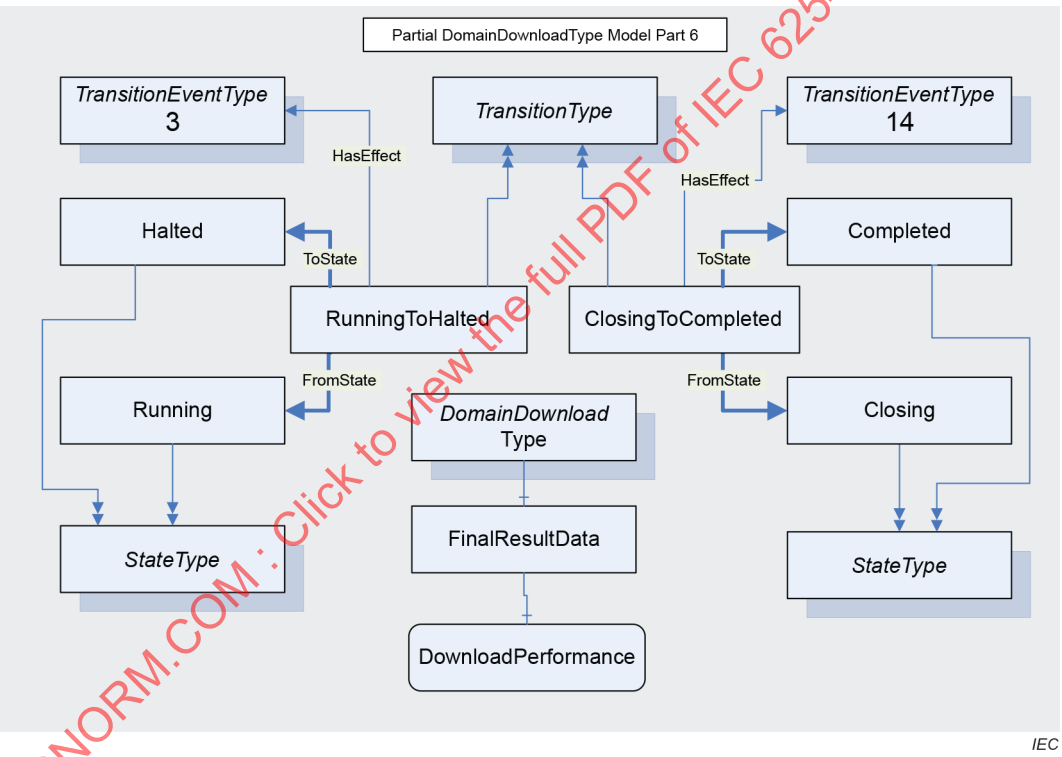


Figure A.10 – Running To Completed model

A.2.6.3 Sequence of operations

Figure A.11 illustrates a normal sequence of service exchanges between a *Client* and *Server* that would occur during the life cycle of a DomainDownloadType *Program Invocation*.

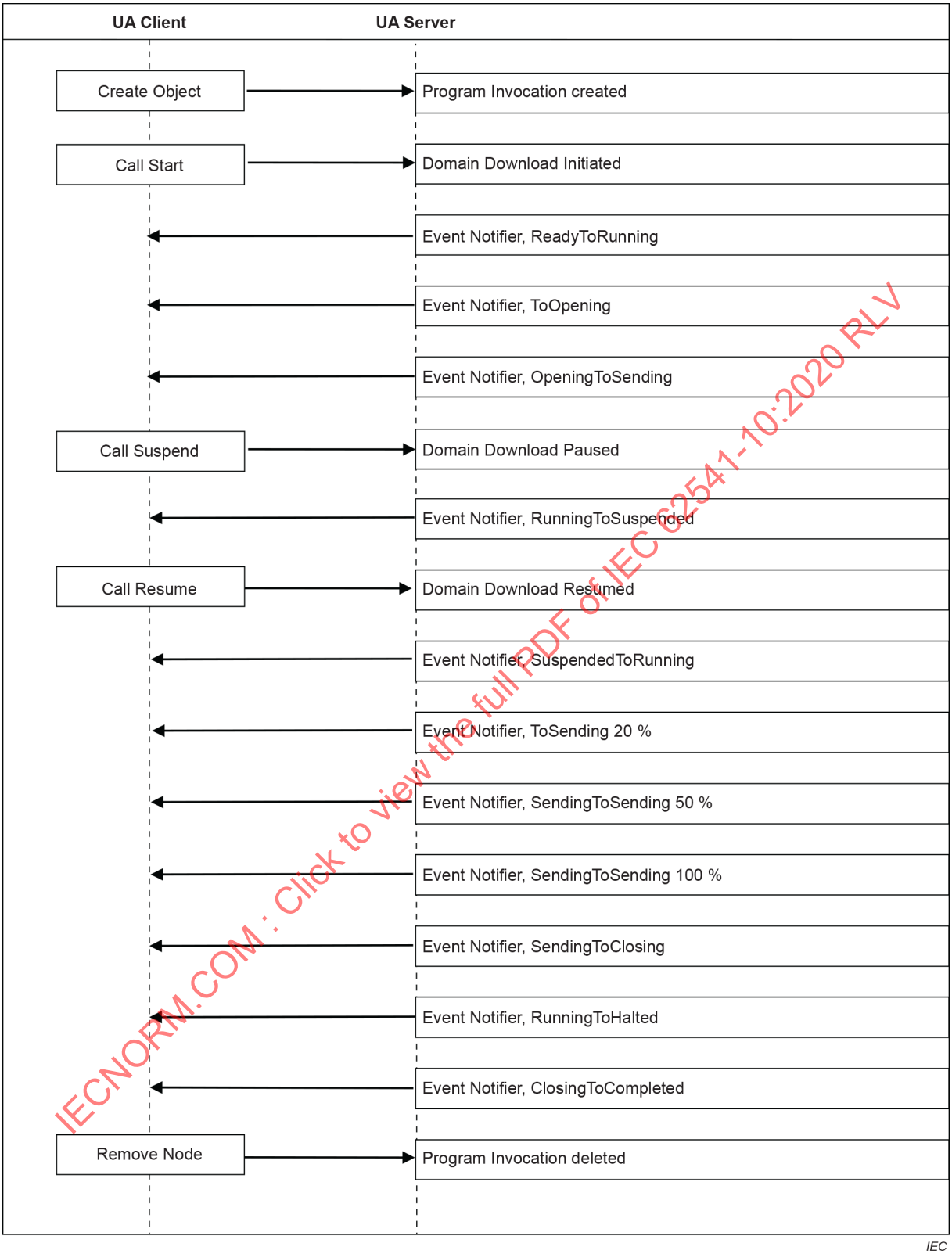


Figure A.11 – Sequence of operations

IEC 62541-10

Edition 3.0 2020-07

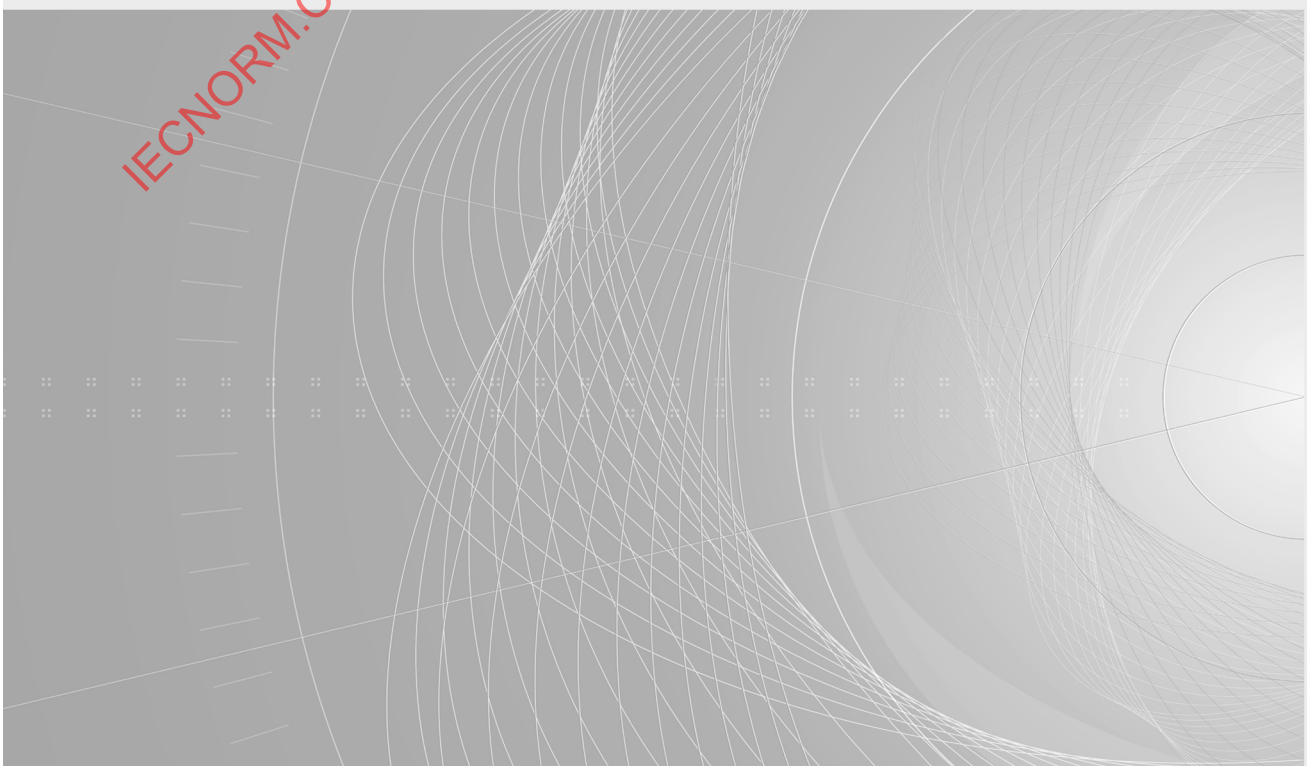
INTERNATIONAL STANDARD

NORME INTERNATIONALE

colour inside

OPC unified architecture – Part 10: Programs

Architecture unifiée OPC – Partie 10: Programmes



IEC 62541-10:2020-07(en-fr)

CONTENTS

INTERNATIONAL ELECTROTECHNICAL COMMISSION

_____

OPC UNIFIED ARCHITECTURE –

Part 10: Programs

FOREWORD

1) The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides (hereafter referred to as "IEC Publication(s)"). Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation. IEC collaborates closely with the International Organization for Standardization (ISO) in accordance with conditions determined by agreement between the two organizations.
2) The formal decisions or agreements of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees.
3) IEC Publications have the form of recommendations for international use and are accepted by IEC National Committees in that sense. While all reasonable efforts are made to ensure that the technical content of IEC Publications is accurate, IEC cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.
4) In order to promote international uniformity, IEC National Committees undertake to apply IEC Publications transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC Publication and the corresponding national or regional publication shall be clearly indicated in the latter.
5) IEC itself does not provide any attestation of conformity. Independent certification bodies provide conformity assessment services and, in some areas, access to IEC marks of conformity. IEC is not responsible for any services carried out by independent certification bodies.
6) All users should ensure that they have the latest edition of this publication.
7) No liability shall attach to IEC or its directors, employees, servants or agents including individual experts and members of its technical committees and IEC National Committees for any personal injury, property damage or other damage of any nature whatsoever, whether direct or indirect, or for costs (including legal fees) and expenses arising out of the publication, use of, or reliance upon, this IEC Publication or any other IEC Publications.
8) Attention is drawn to the Normative references cited in this publication. Use of the referenced publications is indispensable for the correct application of this publication.
9) Attention is drawn to the possibility that some of the elements of this IEC Publication may be the subject of patent rights. IEC shall not be held responsible for identifying any or all such patent rights.

IEC 62541-10 has been prepared by subcommittee 65E: Devices and integration in enterprise systems, of IEC technical committee 65: Industrial-process measurement, control and automation.

This third edition cancels and replaces the second edition published in 2015.

This edition includes several clarifications and in addition the following significant technical changes with respect to the previous edition:

a) Changed ProgramType to ProgramStateMachineType. This is in line with the NodeSet (and thus implementations). In ProgramDiagnosticDataType: changed the definition of lastInputArguments and lastOutputArguments and added two additional fields for the argument values. Also changed StatusResult into StatusCode. Created new version of the type to ProgramDiagnostic2DataType.
b) Changed Optional modelling rule to OptionalPlaceHolder for Program control Methods. Following the clarification in IEC 62541-3, this now allows subtypes (or instances) to add arguments.

The text of this standard is based on the following documents:

FDIS	Report on voting
65E/719/FDIS	65E/735/RVD

Full information on the voting for the approval of this International Standard can be found in the report on voting indicated in the above table.

This document has been drafted in accordance with the ISO/IEC Directives, Part 2.

Throughout this document and the other parts of the IEC 62541 series, certain document conventions are used:

Italics are used to denote a defined term or definition that appears in Clause 3 in one of the parts of the series.

Italics are also used to denote the name of a service input or output parameter or the name of a structure or element of a structure that are usually defined in tables.

The *italicized terms and names* are also, with a few exceptions, written in camel-case (the practice of writing compound words or phrases in which the elements are joined without spaces, with each element's initial letter capitalized within the compound). For example the defined term is *AddressSpace* instead of *Address Space*. This makes it easier to understand that there is a single definition for *AddressSpace*, not separate definitions for Address and Space.

A list of all parts of the IEC 62541 series, published under the general title *OPC Unified Architecture*, can be found on the IEC website.

The committee has decided that the contents of this document will remain unchanged until the stability date indicated on the IEC website under "http://webstore.iec.ch" in the data related to the specific document. At this date, the document will be

- reconfirmed,
- withdrawn,
- replaced by a revised edition, or
- amended.

IMPORTANT – The 'colour inside' logo on the cover page of this publication indicates that it contains colours which are considered to be useful for the correct understanding of its contents. Users should therefore print this document using a colour printer.

OPC UNIFIED ARCHITECTURE –

Part 10: Programs

1 Scope

This part of IEC 62451 defines the *information model* associated with *Programs* in the OPC Unified Architecture. This includes the description of the *NodeClasses,* standard *Properties, Methods* and *Events* and associated behaviour and information for *Programs*.

The complete Address Space model including all *NodeClasses* and *Attributes* is specified in IEC 62541-3. The *Services* such as those used to invoke the *Methods* used to manage *Programs* are specified in IEC 62541-4.

2 Normative references

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

IEC TR 62541-1, *OPC Unified Architecture – Part 1: Overview and Concepts*

IEC 62541-3, *OPC Unified Architecture – Part 3: Address Space Model*

IEC 62541-4, *OPC Unified Architecture – Part 4: Services*

IEC 62541-5, *OPC Unified Architecture – Part 5: Information Model*

IEC 62541-7, *OPC Unified Architecture – Part 7: Profiles*

3 Terms, definitions and abbreviated terms

3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in IEC TR 62541-1, IEC 62541-3 and the following apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

- IEC Electropedia: available at http://www.electropedia.org/
- ISO Online browsing platform: available at http://www.iso.org/obp

3.1.1
function
programmatic task performed by a *Server* or device, usually accomplished by computer code execution

3.1.2
finite state machine
sequence of states and valid state transitions along with the causes and effects of those state transitions that define the actions of a *Program* in terms of discrete stages

3.1.3
ProgramStateMachineType
type definition of a *Program* and is a subtype of the *FiniteStateMachineType*

3.1.4
program control method
Method having specific semantics designed for the control of a *Program* by causing a state transition

3.1.5
program invocation
unique *Object* instance of a *Program* existing on a *Server*

Note 1 to entry: A *Program Invocation* is distinguished from other *Object* instances of the same *ProgramStateMachineType* by the object node's unique browse path.

3.2 Abbreviated terms

DA data access

FSM finite state machine

HMI human–machine interface

UA Unified Architecture

4 Concepts

4.1 General

Integrated automation facilities manage their operations through the exchange of data and the coordinated invocation of system *Functions* as illustrated in Figure 1. *Services* are required to perform the data exchanges and to invoke the *Functions* that constitute system operation. These *Functions* may be invoked through Human Machine Interfaces, cell controllers, or other supervisory control and data acquisition type systems. OPC UA defines *Methods* and *Programs* as an interoperable way to advertise, discover, and request these *Functions*. They provide a normalizing mechanism for the semantic description, invocation, and result reporting of these *Functions*. Together *Methods* and *Programs* complement the other OPC UA *Services* and *ObjectTypes* to facilitate the operation of an automation environment using a client-server hierarchy.

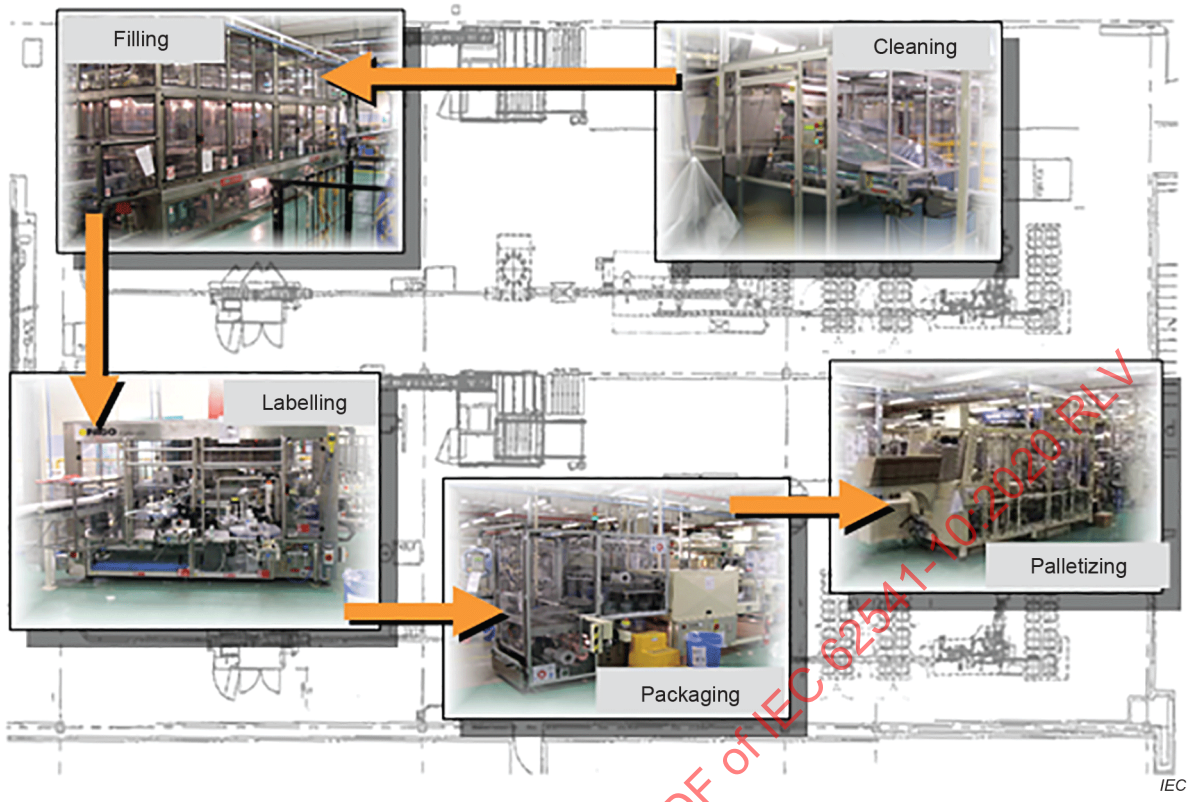


Figure 1 – Automation facility control

Methods and *Programs* model *Functions* typically have different scopes, behaviours, lifetimes, and complexities in *OPC Servers* and the underlying systems. These *Functions* are not normally characterized by the reading or writing of data which is accomplished with the OPC UA A*ttribute* service set.

Methods represent basic *Functions* in the *Server* that can be invoked by a *Client*. *Programs,* by contrast, model more complex and stateful functionality in the system. For example, a method call may be used to perform a calculation or reset a counter. A *Program* is used to run and control a batch process, execute a machine tool part program, or manage a domain download. *Methods* and their invocation mechanism are described in IEC 62541-3 and IEC 62541-4.

This document describes the extensions to, or specific use of, the core capabilities defined in IEC 62541-5 as required for *Programs*.

4.2 Programs

4.2.1 Overview

Programs are complex *Functions* in a *Server* or underlying system that can be invoked and managed by a *Client*. *Programs* can represent any level of functionality within a system or process in which *Client* control or intervention is required and progress monitoring is desired. Figure 2 illustrates the model.

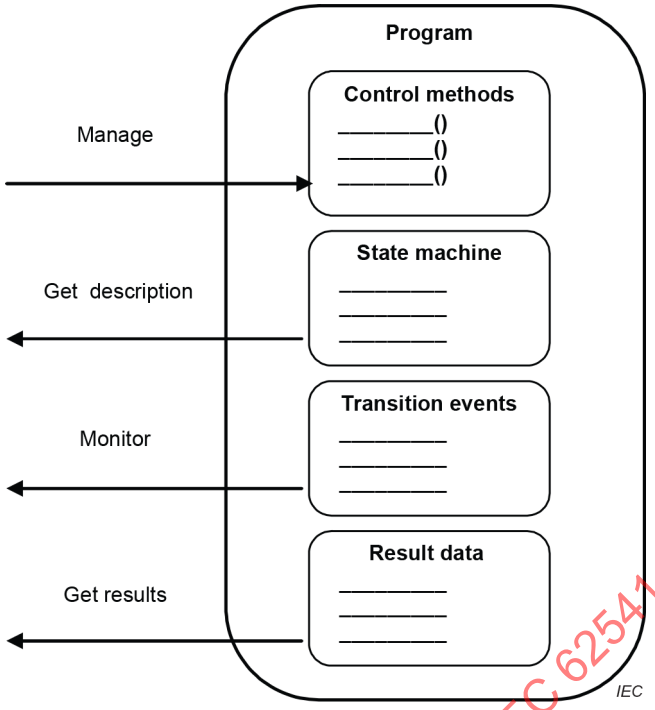


Figure 2 – Program illustration

Programs are stateful and transition through a prescribed sequence of states as they execute. Their behaviour is defined by a *Program Finite State Machine (PFSM)*. The elements of the PFSM describe the phases of a *Program's* execution in terms of valid transitions between a set of states, the stimuli or causes of those transitions, and the resultant effects of the transitions.

4.2.2 Security considerations

Since *Programs* can be used to perform advanced control algorithms or other actions, their use should be restricted to personnel with appropriate access rights. It is recommended that *AuditUpdateMethodEvents* are generated to allow monitoring the number of running *Programs* in addition to their execution frequency.

4.2.3 Program Finite State Machine

The states, transitions, causes and effects that compose the *Program Finite State Machine* are listed in Table 1 and illustrated in Figure 3.

Table 1 – Program Finite State Machine

No.	Transition name	Cause	From state	To state	Effect
1	HaltedToReady	Reset Method	Halted	Ready	Report Transition 1 Event/Result
2	ReadyToRunning	Start Method	Ready	Running	Report Transition 2 Event/Result
3	RunningToHalted	Halt Method or Internal (Error)	Running	Halted	Report Transition 3 Event/Result
4	RunningToReady	Internal	Running	Ready	Report Transition 4 Event/Result
5	RunningToSuspended	Suspend Method	Running	Suspended	Report Transition 5 Event/Result
6	SuspendedToRunning	Resume Method	Suspended	Running	Report Transition 6 Event/Result
7	SuspendedToHalted	Halt Method	Suspended	Halted	Report Transition 7 Event/Result
8	SuspendedToReady	Internal	Suspended	Ready	Report Transition 8 Event/Result
9	ReadyToHalted	Halt Method	Ready	Halted	Report Transition 9 Event/Result

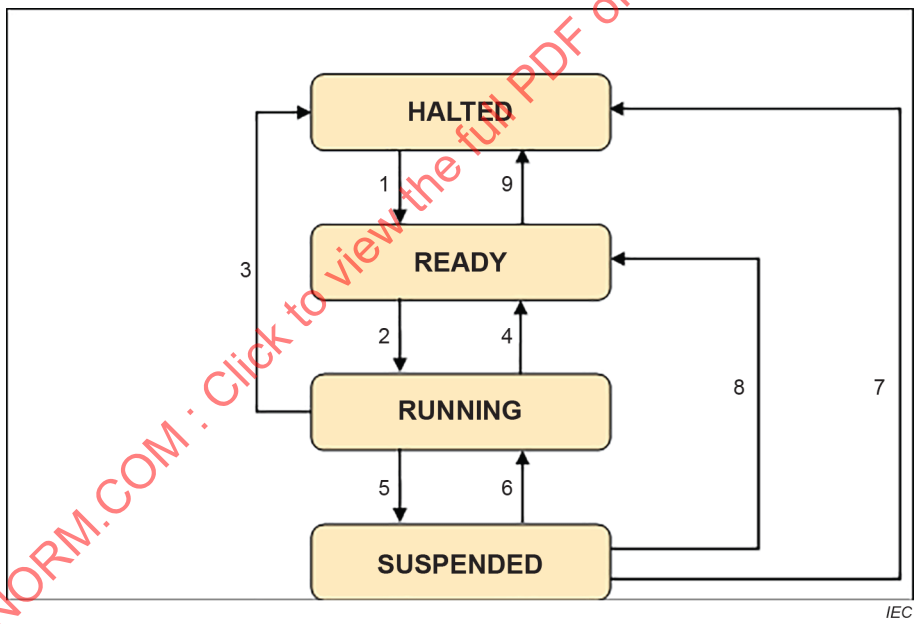


Figure 3 – Program states and transitions

4.2.4 Program states

A standard set of base states is defined for *Programs* as part of the *Program Finite State Machine.* These states represent the stages in which a *Program* can exist at an instant in time as viewed by a *Client*. This state is the *Program's* current state. All *Programs* shall support this base set. A *Program* may or may not require a *Client* action to cause the state to change. The states are formally defined in Table 2.

Table 2 – Program states

State	Description
Ready	The *Program* is properly initialized and may be started.
Running	The *Program* is executing making progress towards completion.
Suspended	The *Program* has been stopped prior to reaching a terminal state but may be resumed.
Halted	The *Program* is in a terminal or failed state, and it cannot be started or resumed without being reset.

The set of states defined to describe a *Program* can be expanded. *Program* sub states can be defined for the base states to provide more resolution of a process and to describe the cause and effect(s) of additional stimuli and transitions. Standards bodies and industry groups may extend the base *Program Finite State Model* to conform to various industry models. For example, the Halted state can include the sub states "Aborted" and "Completed" to indicate if the *Function* achieved a successful conclusion prior to the transition to Halted. Transitional states such as "Starting" or "Suspending" might also be extensions of the Running state, for example.

4.2.5 State transitions

A standard set of state transitions is defined for the *Program Finite State Machine*. These transitions define the valid changes to the *Program's* current state in terms of an initial state and a resultant state. The transitions are formally defined in Table 3.

Table 3 – Program state transitions

Transition no.	Transition name	Initial state	Resultant state
1	HaltedToReady	Halted	Ready
2	ReadyToRunning	Ready	Running
3	RunningToHalted	Running	Halted
4	RunningToReady	Running	Ready
5	RunningToSuspended	Running	Suspended
6	SuspendedToRunning	Suspended	Running
7	SuspendedToHalted	Suspended	Halted
8	SuspendedToReady	Suspended	Ready
9	ReadyToHalted	Ready	Halted

4.2.6 Program state transition stimuli

The stimuli or causes for a *Program's* state transitions can be internal to the *Server* or external. The completion of machining steps, the detection of an alarm condition, or the transmission of a data packet are examples of internal stimuli. *Methods* are an example of external stimuli. Standard *Methods* are defined which act as stimuli for the control of a *Program*.

4.2.7 Program Control Methods

Clients manage a *Program* by calling *Methods*. The *Methods* impact a *Program's* behaviour by causing specified state transitions. The state transitions dictate the actions performed by the *Program*. This document defines a set of standard *Program Control Methods*. These *Methods* provide sufficient means for a *Client* to run a *Program*.

Table 4 lists the set of defined *Program Control Methods*. Each *Method* causes transitions from specified states and shall be called when the *Program* is in one of those states.

Individual *Programs* can optionally support any subset of the *Program Control Methods*. For example, some *Programs* may not be permitted to suspend and so would not provide the *Suspend* and *Resume Methods*.

Programs can support additional user defined *Methods*. User defined *Methods* shall not change the behaviour of the base *Program Finite State Machine*.

Table 4 – Program Control Methods

Method Name	Description
Start	Causes the *Program* to transition from the Ready state to the Running state.
Suspend	Causes the *Program* to transition from the Running state to the Suspended state.
Resume	Causes the *Program* to transition from the Suspended state to the Running state.
Halt	Causes the *Program* to transition from the Ready, Running or Suspended state to the Halted state.
Reset	Causes the *Program* to transition from the Halted state to the Ready state.

All *Program Control Methods* are defined with their *BrowseName* on the *ProgramStateMachineType* with the *OptionalPlaceholder ModellingRule*. As defined in IEC 62541-3, this rule allows the inclusion of *Arguments* to these Methods on subtypes or on instances. For example, a Start *Method* may include an options argument that specifies dynamic options used to determine some program behaviour. The *Method Call* service specified in IEC 62541-4 defines a return status. This return status indicates the success of the *Program Control Method* or a reason for its failure.

4.2.8 Program state transition effects

A *Program's* state transition generally has a cause and also yields an effect. The effect is a by product of a *Program* state transition that can be used by a *Client* to monitor the progress of the *Program*. Effects can be internal or external. An external effect of a state transition is the generation of an *Event* notification. Each *Program* state transition is associated with a unique *Event*. These *Events* reflect the progression and trajectory of the *Program* through its set of defined states. The internal effects of a state transition can be the performance of some programmatic action such as the generation of data.

4.2.9 Program result data

4.2.9.1 Overview

Result data is generated by a running *Program*. The result data can be intermediate or final. Result data may be associated with specific *Program* state transitions.

4.2.9.2 Intermediate result data

Intermediate result data is transient and is generated by the *Program* in conjunction with non-terminal state transitions. The data items that compose the intermediate results are defined in association with specific *Program* state transitions. Their values are relevant only at the transition level.

Each *Program* state transition can be associated with different result data items. Alternately, a set of transitions can share a result data item. Percentage complete is an example of intermediate result data. The value of percentage complete is produced when the state transition occurs and is available to the *Client*.

Clients acquire intermediate result data by subscribing to *Program* state transition *Events*. The *Events* specify the data items for each transition. When the transition occurs, the generated *Event* conveys the result data values captured to the subscribed *Clients*. If no *Client* is monitoring the *Program*, intermediate result data may be discarded.

4.2.9.3 Terminal result data

Terminal result data is the final data generated by the *Program* as it ceases execution. Total execution time, number of widgets produced, and fault condition encountered are examples of terminal result data. When the *Program* enters the terminal state, this result data can be conveyed to the *Client* by the transition *Event*. Terminal result data is also available within the *Program* to be read by a *Client* after the program stops. This data persists until the *Program* Instance is rerun or deleted.

4.2.9.4 Monitoring Programs

Clients can monitor the activities associated with a *Program's* execution. These activities include the invocation of the management *Methods*, the generation of result data, and the progression of the *Program* through its states. *Audit Events* are provided for *Method Calls* and state transitions. These *Events* allow a record to be maintained of the *Clients* that interacted with any *Program* and the *Program* state transitions that resulted from that interaction.

4.2.10 Program lifetime

4.2.10.1 Overview

Programs can have different lifetimes. Some *Programs* may always be present on a *Server* while others are created and removed. Creation and removal can be controlled by a *Client* or may be restricted to local means.

A *Program* can be *Client* creatable. If a *Program* is *Client* creatable, then the *Client* can add the *Program* to the *Server*. The *Object Create Method* defined in IEC 62541-3 is used to create the *Program* instance. The initial state of the *Program* can be Halted or Ready. Some *Programs*, for example, may require that a resource becomes available after its creation and before it is ready to run. In this case, it would be initialized in the Halted state and transition to Ready when the resource is delivered.

A *Program* can be *Client* removable. If the *Program* is *Client* removable, then the *Client* can delete the *Program* instance from the *Server*. The *DeleteNodes Service* defined in IEC 62541-4 is used to remove the *Program* Instance. The *Program* shall be in a Halted state to be removed. A *Program* may also be auto removable. An auto removable *Program* deletes itself when execution has terminated.

4.2.10.2 Program instances

Programs can be multiple instanced or single instanced. A *Server* can support multiple instances of a *Program* if these *Program* Instances can be run in parallel. For example, the *Program* may define a *Start Method* that has an input argument to specify which resource is acted upon by its *Functions*. Each instance of the *Program* is then started designating use of different resources. The *Client* can discover all instances of a *Program* that are running on a *Server*. Each instance of a *Program* is uniquely identified on the *Server* and is managed independently by the *Client*.

4.2.10.3 Program recycling

Programs can be run once or run multiple times (recycled). A *Program* that is run once will remain in the Halted state indefinitely once it has run. The normal course of action would be to delete it following the inspection of its terminal results.

Recyclable *Programs* may have a limited or unlimited cycle count. These *Programs* may require a reset step to transition from the Halted state to the Ready state. This allows for replenishing resources or reinitializing parameters prior to restarting the *Program*. The *Program Control Method* "*Reset*" triggers this state transition and any associated actions or effects.

5 Model

5.1 General

The *Program* model extends the *FiniteStateMachineType and* basic *ObjectType* models presented in IEC 62541-5. Each *Program* has a *Type Definition* that is the subtype of the *FiniteStateMachineType.* The *ProgramStateMachineType* describes the *Finite State Machine* model supported by any *Program Invocation* of that type. The *ProgramStateMachineType* also defines the property set that characterizes specific aspects of that *Program's* behaviour such as lifetime and recycling as well as specifying the result data that is produced by the *Program*.

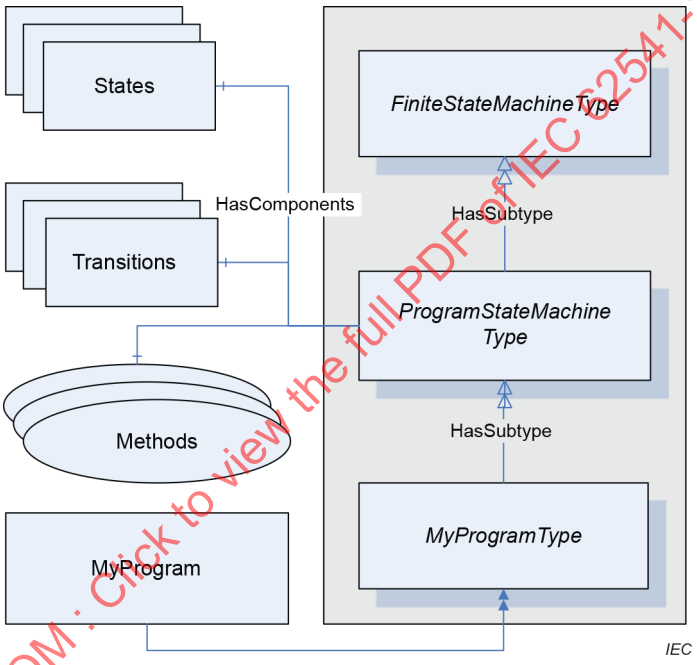


Figure 4 – Program Type

The base *ProgramStateMachineType* defines the standard *Finite State Machine* specified for all *Programs*. This includes the states, transitions, and transition causes (*Methods*) and effects (*Events*). Subtypes of the base *ProgramStateMachineType* can be defined to extend or more specifically characterize the behaviour of an individual *Program* as illustrated with "MyProgramType" in Figure 4.

5.2 ProgramStateMachineType

5.2.1 Overview

The additional properties and components that compose the *ProgramStateMachineType* are listed in Table 5. No *ProgramStateMachineType* specific semantics are assigned to the other base *ObjectType* or *FiniteStateMachineType Attributes* or *Properties*.

Table 5 – ProgramStateMachineType

Attribute	Value				
	Includes all attributes specified for the FiniteStateMachineType				
BrowseName	ProgramStateMachineType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**Data Type**	**TypeDefinition**	**Modelling Rule**
HasProperty	Variable	Creatable	Boolean	PropertyType	–
HasProperty	Variable	Deletabe	Boolean	PropertyType	Mandatory
HasProperty	Variable	AutoDelete	Boolean	PropertyType	Mandatory
HasProperty	Variable	RecycleCount	Int32	PropertyType	Mandatory
HasProperty	Variable	InstanceCount	UInt32	PropertyType	–
HasProperty	Variable	MaxInstanceCount	UInt32	PropertyType	–
HasProperty	Variable	MaxRecycleCount	UInt32	PropertyType	–
HasComponent	Variable	ProgramDiagnostic	ProgramDiagnostic2DataType	ProgramDiagnostic2Type	Optional
HasComponent	Object	Halted		StateType	–
HasComponent	Object	Ready		StateType	–
HasComponent	Object	Running		StateType	–
HasComponent	Object	Suspended		StateType	–
HasComponent	Object	HaltedToReady		TransitionType	–
HasComponent	Object	ReadyToRunning		TransitionType	–
HasComponent	Object	RunningToHalted		TransitionType	–
HasComponent	Object	RunningToReady		TransitionType	–
HasComponent	Object	RunningToSuspended		TransitionType	–
HasComponent	Object	SuspendedToRunning		TransitionType	–
HasComponent	Object	SuspendedToHalted		TransitionType	–
HasComponent	Object	SuspendedToReady		TransitionType	–
HasComponent	Object	ReadyToHalted		TransitionType	–
HasComponent	Method	Start			OptionalPlaceholder
HasComponent	Method	Suspend			OptionalPlaceholder
HasComponent	Method	Reset			OptionalPlaceholder
HasComponent	Method	Halt			OptionalPlaceholder
HasComponent	Method	Resume			OptionalPlaceholder
HasComponent	Object	FinalResultData		BaseObjectType	Optional

5.2.2 ProgramStateMachineType Properties

The *Creatable Property* is a boolean that specifies if *Program Invocations* of this *ProgramStateMachineType* can be created by a *Client*. If False, these *Program Invocations* are persistent or may only be created by the *Server*.

The *Deletable Property* is a boolean that specifies if a *Program Invocation* of this *ProgramStateMachineType* can be deleted by a *Client*. If False, these *Program Invocations* can only be deleted by the *Server*.

The *AutoDelete Property* is a boolean that specifies if *Program Invocations* of this *ProgramStateMachineType* are removed by the *Server* when execution terminates. If False, these *Program Invocations* persist on the *Server* until they are deleted by the *Client*. When the *Program Invocation* is deleted, any result data associated with the instance is also removed.

The *RecycleCount Property* is an unsigned integer that specifies the number of times a *Program Invocation* of this type has been recycled or restarted from its starting point (not resumed). Note that the Reset Method may be required to prepare a *Program* to be restarted.

The *MaxRecycleCount Property* is an integer that specifies the maximum number of times a *Program Invocation* of this type can be recycled or restarted from its starting point (not resumed). If the value is less than 0, then there is no limit to the number of restarts. If the value is zero, then the Program may not be recycled or restarted.

The *InstanceCount Property* is an unsigned integer that specifies the number of *Program Invocations* of this type that currently exist.

The *MaxInstanceCount Property* is an integer that specifies the maximum number of *Program Invocations* of this type that can exist simultaneously on this *Server*. If the value is less than 0, then there is no limit.

5.2.3 ProgramStateMachineType components

5.2.3.1 Overview

The *ProgramStateMachineType* components consist of a set of *References* to the *Object* instances of *StateTypes*, *TransitionTypes*, *EventTypes* and the *Methods* that collectively define the *Program FiniteStateMachine*.

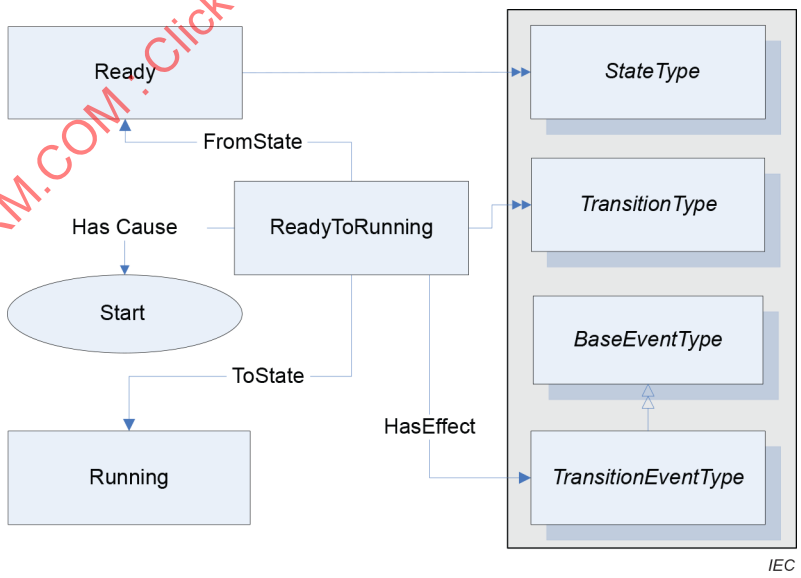


Figure 5 – Program FSM References

Figure 5 illustrates the component *References* that define the associations between two of the *ProgramStateMachineType's* states, Ready and Running. The complementary ReferenceTypes have been omitted to simplify the illustration.

5.2.3.2 ProgramStateMachineType states

Table 6 specifies the *ProgramStateMachineType's* state *Objects*. These *Objects* are instances of the *StateType* defined in IEC 62541-5. Each state is assigned a unique *StateNumber* value. Subtypes of the *ProgramStateMachineType* can add references from any state to a subordinate or nested *StateMachine Object* to extend the *FiniteStateMachine*.

Table 6 – Program states

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	NOTES
States					
Halted	HasProperty	StateNumber	11	PropertyType	
	ToTransition	HaltedToReady		TransitionType	
	FromTransition	RunningToHalted		TransitionType	
	FromTransition	SuspendedToHalted		TransitionType	
	FromTransition	ReadyToHalted		TransitionType	
Ready	HasProperty	StateNumber	12	PropertyType	
	FromTransition	HaltedToReady		TransitionType	
	ToTransition	ReadyToRunning		TransitionType	
	FromTransition	RunningToReady		TransitionType	
	ToTransition	ReadyToHalted		TransitionType	
Running	HasProperty	StateNumber	13	PropertyType	
	ToTransition	RunningToHalted		TransitionType	
	ToTransition	RunningToReady		TransitionType	
	ToTransition	RunningToSuspended		TransitionType	
	FromTransition	ReadyToRunning		TransitionType	
	FromTransition	SuspendedToRunning		TransitionType	
Suspended	HasProperty	StateNumber	14	PropertyType	
	ToTransition	SuspendedToRunning		TransitionType	
	ToTransition	SuspendedToHalted		TransitionType	
	ToTransition	SuspendedToReady		TransitionType	
	FromTransition	RunningToSuspended		TransitionType	

The Halted state is the idle state for a *Program*. It can be an initial state or a terminal state. As an initial state, the *Program Invocation* cannot begin execution due to conditions at the *Server*. As a terminal state, Halted can indicate either a failed or completed *Program*. A subordinate state or result can be used to distinguish the nature of the termination. The Halted state references four *Transition Objects*, which identify the allowed state transitions to the Ready state and from the Ready, Running, and Suspended states.

The Ready state indicates that the *Program* is prepared to begin execution. *Programs* that are ready to begin upon their creation may transition immediately to the Ready state. The Ready state references four *Transition Objects,* which identify the allowed state transitions to the Running and Halted states and from the Halted and Ready states.

The Running state indicates that the *Program* is actively performing its *Function*. The Running state references five *Transition Objects,* which identify the allowed state transitions to the Halted, Ready, and Suspended states and from the Ready and Suspended states.

The Suspended state indicates that the *Program* has stopped performing its *Function*, but retains the ability to resume the *Function* at the point at which it was executing when suspended. The Suspended state references four *Transition Objects*, which identify the allowed state transitions to the Ready, Running, and Halted state and from the Ready state.

5.2.3.3 ProgramStateMachineType transitions

ProgramStateMachineType Transitions are instances of the *TransitionType* defined in IEC 62541-5 which also includes the definitions of the ToState, FromState, HasCause, and HasEffect references used. Table 7 specifies the transitions defined for the *ProgramStateMachineType*. Each transition is assigned a unique *TransitionNumber*. The Notes column indicates when a cause is referencing *Methods* and when effects are optional.

Table 7 – Program transitions

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	Notes
Transitions					
HaltedToReady	HasProperty	TransitionNumber	1	PropertyType	
	ToState	Ready		StateType	
	FromState	Halted		StateType	
	HasCause	Reset			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
ReadyToRunning	HasProperty	TransitionNumber	2	PropertyType	
	ToState	Running		StateType	
	FromState	Ready		StateType	
	HasCause	Start			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
RunningToHalted	HasProperty	TransitionNumber	3	PropertyType	
	ToState	Halted		StateType	
	FromState	Running		StateType	
	HasCause	Halt			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
RunningToReady	HasProperty	TransitionNumber	4	PropertyType	
	ToState	Ready		StateType	
	FromState	Runnning		StateType	
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	Notes
RunningToSuspended	HasProperty	TransitionNumber	5	PropertyType	
	ToState	Running		StateType	
	FromState	Suspended		StateType	
	HasCause	Suspend			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
SuspendedToRunning	HasProperty	TransitionNumber	6	PropertyType	
	ToState	Running		StateType	
	FromState	Suspended		StateType	
	HasCause	Resume			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
SuspendedToHalted	HasProperty	TransitionNumber	7	PropertyType	
	ToState	Halted		StateType	
	FromState	Suspended		StateType	
	HasCause	Halt			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
SuspendedToReady	HasProperty	TransitionNumber	8	PropertyType	
	ToState	Ready		StateType	
	FromState	Suspended		StateType	
	HasCause	Reset			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
ReadyToHalted	HasProperty	TransitionNumber	9	PropertyType	
	ToState	Halted		StateType	
	FromState	Ready		StateType	
	HasCause	Halt			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			

The *HaltedToReady* transition specifies the transition from the Halted to Ready states. It may be caused by the *Reset Method.*

The *ReadyToRunning* transition specifies the transition from the Ready to Running states. It is caused by the *Start Method.*

The *RunningToHalted* transition specifies the transition from the Running to Halted states. It is caused by the *Halt Method.*

The *RunningToReady* transition specifies the transition from the Running to Ready states.

The *RunningToSuspended* transition specifies the Transition from the Running to Suspended states. It is caused by the *Suspend Method.*

The *SuspendedToRunning* transition specifies the transition from the Suspended to Running states. It is caused by the *Resume Method.*

The *SuspendedToHalted* transition specifies the transition from the Suspended to Halted states. It is caused by the *Halt Method.*

The *SuspendedToReady* transition specifies the transition from the Suspended to Ready states. It is caused internally.

The *ReadyToHalted* transition specifies the transition from the Ready to Halted states. It is caused by the *Halt Method.*

Two *HasEffect* references are specified for each *Program* transition. These effects are *Events* of *ProgramTransitionEventType* and *AuditProgramTransitionEventType* defined in 5.2.5. The *ProgramTransitionEventType* notifies *Clients* of the *Program* transition and conveys result data. The *AuditProgramTransitionEventType* is used to audit transitions that result from *Program Control Methods*. The "*Audit Server Facet*" *Profile* defined in IEC 62541-7 requires support of the *AuditProgramTransitionEventType*.

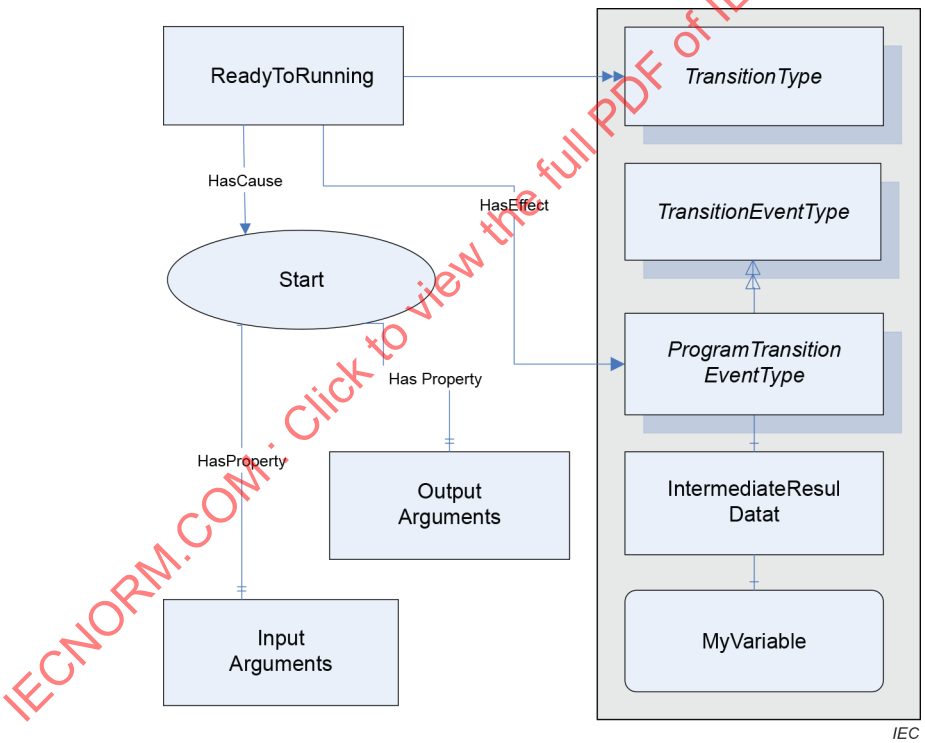


Figure 6 – ProgramStateMachineType causes and effects

5.2.4 ProgramStateMachineType causes (Methods)

5.2.4.1 Overview

The *ProgramStateMachineType* includes references to the Causes of specific *Program* state transitions. These causes refer to *Method* instances. *Programs* that do not support *a Program Control Method* will omit the Causes reference to that *Method* from the *ProgramStateMachineType* references. If a *Method's Causes* reference is omitted from the *ProgramStateMachineType* then a *Client* cannot cause the associated state transition. The *Method* instances referenced by the *ProgramStateMachineType* identify the *InputArguments*

and *OutputArguments* required for the *Method* calls to *Program Invocations* of that *ProgramStateMachineType*. Table 8 specifies the *Methods* defined as Causes for *ProgramStateMachineTypes*. Figure 6 illustrates the *References* associating the components and *Properties* of *Methods* and *Events* with Program transitions.

Table 8 – ProgramStateMachineType causes

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	NOTES
Causes					
Start	HasProperty	InputArguments		PropertyType	Optional
	HasProperty	OutputArguments		PropertyType	Optional
Suspend	HasProperty	InputArguments		PropertyType	Optional
	HasProperty	OutputArguments		PropertyType	Optional
Resume	HasProperty	InputArguments		PropertyType	Optional
	HasProperty	OutputArguments		PropertyType	Optional
Halt	HasProperty	InputArguments		PropertyType	Optional
	HasProperty	OutputArguments		PropertyType	Optional
Reset	HasProperty	InputArguments		PropertyType	Optional
	HasProperty	OutputArguments		PropertyType	Optional

The *Start Method* causes the *ReadyToRunning Program* transition.

The *Suspend Method* causes the *RunningToSuspended Program* transition.

The *Resume Method* causes the *SuspendedToRunning Program* transition.

The *Halt Method* causes the *RunningToHalted, SuspendedToHalted, or ReadyToHalted Program* transition depending on the current state of the *Program*.

The *Reset Method* causes the *HaltedToReady Program* transition.

5.2.4.2 Standard attributes

The *Executable Method* attribute indicates if a method can currently be executed. For *Program Control Methods*, this means that the owning *Program* has a current state that supports the transition caused by the *Method.*

5.2.4.3 Standard properties

Methods can reference a set of *InputArguments.* For each *ProgramStateMachineType*, a set of *InputArguments* may be defined for the supported *Program Control Methods*. The data passed in the arguments supplements the information required by the *Program* to perform its *Function*. All calls to a *Program Control Method* for each *Program Invocation* of that *ProgramStateMachineType* shall pass the specified arguments.

Methods can reference a set of *OutputArguments*. For each *ProgramStateMachineType*, a set of *OutputArguments* is defined for the supported *Program Control Methods*. All calls to a *Program Control Method* for each *Program Invocation* of that *ProgramStateMachineType* shall pass the specified arguments.

5.2.5 ProgramStateMachineType effects (Events)

5.2.5.1 Overview

The *ProgramStateMachineType* includes component references to the Effects of each of the *Program's* state transitions. These Effects are *Events.* Each *Transition* shall have a *HasEffect Reference* to a *ProgramTransitionEventType* and can have an *AuditProgramTransitionEventType.* When the transition occurs, *Event* notifications of the referenced type are generated for subscribed *Clients.* The *Program Invocation* may serve as the *EventNotifier* for these *Events* or an owning *Object* or the *Server Object* may provide the notifications.

ProgramTransitionEventTypes provide the means for delivering result data and confirming state transitions for subscribed *Clients* on each defined *Program State Transition.* The *AuditProgramTransitionEventType* allows the auditing of changes to the *Program*'s state in conjunction with *Client Method Calls.*

5.2.5.2 ProgramTransitionEventType

The *ProgramTransitionEventType* is a subtype of the *TransitionEventType*. It is used with *Programs* to acquire intermediate or final results or other data associated with a state transition. A *Program* can have a unique *ProgramTransitionEventType* definition for any transition. Each *ProgramTransitionEventType* specifies the *IntermediateResult* data specific to the designated state transition on that *ProgramStateMachineType*. Each transition can yield different intermediate result data. Table 9 specifies the *ProgramTransitionEventType*.

Table 10 identifies the *ProgramTransitionEventTypes* that are specified for *ProgramStateMachineTypes*.

Table 9 – ProgramTransitionEventType

Attribute	Value				
BrowseName	ProgramTransitionEventType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the base *TransitionEventType* defined in IEC 62541-5.					
HasComponent	Object	IntermediateResult		BaseObjectType	Optional

TransitionNumber identifies the *Program* transition that triggered the *Event*.

FromStateNumber identifies the state before the *Program* transition.

ToStateNumber identifies the state after the *Program* transition.

The *IntermediateResult* is an *Object* that aggregates a set of *Variables* whose values are relevant for the *Program* at the instant of the associated transition. The *ObjectType* for the *IntermediateResult* specifies the collection of *Variables* using a set of *HasComponent References*.

Table 10 – ProgramTransitionEvents

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	Notes
Effects					
HaltedToReadyEvent					
	HasProperty	TransitionNumber	1	PropertyType	
	HasProperty	FromStateNumber	1	PropertyType	
	HasProperty	ToStateNumber	2	PropertyType	
	HasComponent	IntermediateResults		ObjectType	Optional
ReadyToRunningEvent					
	HasProperty	TransitionNumber	2	PropertyType	
	HasProperty	FromStateNumber	2	PropertyType	
	HasProperty	ToStateNumber	3	PropertyType	
	HasComponent	IntermediateResults		ObjectType	Optional
RunningToHaltedEvent					
	HasProperty	TransitionNumber	3	PropertyType	
	HasProperty	FromStateNumber	3	PropertyType	
	HasProperty	ToStateNumber	1	PropertyType	
	HasComponent	IntermediateResults		ObjectType	Optional
RunningToReadyEvent					
	HasProperty	TransitionNumber	4	PropertyType	
	HasProperty	FromStateNumber	3	PropertyType	
	HasProperty	ToStateNumber	2	PropertyType	
	HasComponent	IntermediateResults		ObjectType	Optional
RunningToSuspendedEvent					
	HasProperty	TransitionNumber	5	PropertyType	
	HasProperty	FromStateNumber	3	PropertyType	
	HasProperty	ToStateNumber	4	PropertyType	
	HasComponent	IntermediateResults		ObjectType	Optional
SuspendedToRunningEvent					
	HasProperty	TransitionNumber	6	PropertyType	
	HasProperty	FromStateNumber	4	PropertyType	
	HasProperty	ToStateNumber	3	PropertyType	
	HasComponent	IntermediateResults		ObjectType	Optional
SuspendedToHaltedEvent					
	HasProperty	TransitionNumber	7	PropertyType	
	HasProperty	FromStateNumber	4	PropertyType	
	HasProperty	ToStateNumber	1	PropertyType	
	HasComponent	IntermediateResults		ObjectType	Optional

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	Notes
SuspendedToReadyEvent					
	HasProperty	TransitionNumber	8	PropertyType	
	HasProperty	FromStateNumber	4	PropertyType	
	HasProperty	ToStateNumber	2	PropertyType	
	HasComponent	IntermediateResults		ObjectType	Optional
ReadyToHaltedEvent					
	HasProperty	TransitionNumber	9	PropertyType	
	HasProperty	FromStateNumber	2	PropertyType	
	HasProperty	ToStateNumber	1	PropertyType	
	HasComponent	IntermediateResults		ObjectType	Optional

5.2.6 AuditProgramTransitionEventType

The *AuditProgramTransitionEventType* is a subtype of the *AuditUpdateStateEventType*. It is used with *Programs* to provide a means to audit the *Program State* transitions associated with any *Client* invoked *Program Control Method*. *Servers* shall generate *AuditProgramTransitionEvents* if they support the *Audit Server Facet Profile* defined in IEC 62541-7.

Table 11 specifies the definition of the *AuditProgramTransitionEventType*.

Table 11 – AuditProgramTransitionEventType

Attribute	Value				
BrowseName	AuditProgramTransitionEventType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *AuditUpdateStateEventType* defined in IEC 62541-5.					
HasProperty	Variable	TransitionNumber	UInt32	PropertyType	Mandatory

This *EventType* inherits all *Properties* of the *AuditUpdateStateEventType* defined in IEC 62541-5, except as noted below.

The *Status Property*, specified in IEC 62541-5, identifies whether the state transition resulted from a Program Control Method call (set *Status* to TRUE) or not (set *Status* to FALSE).

The *SourceName*, specified in IEC 62541-5, identifies the *Method* causing the *Program* transition when it is the result of a *Client* invoked *ProgramControlMethod*. The *SourceName* is prefixed with "Method/" and the name of the *ProgramControlMethod*, "Method/Start" for example.

The *ClientUserId Property*, specified in IEC 62541-5, identifies the user of the *Client* that issued the *Program Control Method* if it is associated with this *Program* state transition.

The *ActionTimeStamp Property*, specified in IEC 62541-5 "AuditEventType", identifies when the time the *Program* state transition that resulted in the *Event* being generated occurred.

The *TransitionNumber Property* is a *Variable* that identifies the transition that triggered the *Event*.

5.2.7 FinalResultData

The *FinalResultData ObjectType* specifies the *VariableTypes* that are preserved when the *Program* has completed its *Function*. The *ObjectType* includes *a HasComponent* for a *VariableType* of each *Variable* that comprises the final result data.

5.2.8 ProgramDiagnostic2 DataType

This structure contains elements that chronicle the *Program Invocation's* activity and can be used to aid in the diagnosis of *Program* problems.

NOTE The original ProgramDiagnosticDataType had flaws. To avoid collisions with existing implementations, a new version with name ProgramDiagnostic2DataType has been created.

Its composition is defined in Table 12.

Table 12 – ProgramDiagnostic2DataType structure

Name	Type	Description
ProgramDiagnostic2DataType	structure	
createSessionId	NodeId	The *CreateSessionId* contains the *SessionId* of the *Session*on which the call to the *Create Method* was issued to create the *Program Invocation*.
createClientName	String	The *CreateClientName* is the name of the *Client*of the *Session*that created the *Program Invocation*.
invocationCreationTime	UtcTime	The *InvocationCreationTime* identifies the time the *Program Invocation* was created.
lastTransitionTime	UtcTime	The *LastTransitionTime* identifies the time of the last *Program* state transition that occurred.
lastMethodCall	String	The *LastMethodCall* identifies the last *Program Method* called on the *Program Invocation*.
lastMethodSessionId	NodeId	The *LastMethodSessionId* contains the *SessionId* of the *Session*on which the last *Program Control Method* call to the *Program Invocation* was issued.
lastMethodInputArguments	Argument[]	The *LastMethodInputArguments* provides the input arguments on the last *Program Method* call.
lastMethodOutputArguments	Argument[]	The *LastMethodOutputArguments* provides the output arguments on the last *Program Method* call.
lastMethodInputValues	BaseDataType[]	The *LastMethodInputValues* preserves the values of the input arguments on the last *Program Method* call. The size and order of this list matches the size and order of the *lastMethodInputArguments* field.
lastMethodOutputValues	BaseDataType[]	The *LastMethodOutputValues* preserves the values of the output arguments on the last *Program Method* call. The size and order of this list matches the size and order of the *lastMethodOutputArguments* field.
lastMethodCallTime	UtcTime	The *LastMethodCallTime* identifies the time of the last Method call to the *Program Invocation*.
lastMethodReturnStatus	StatusCode	The *LastMethodReturnStatus* preserves the value of the return status for the last *Program Control Method* requested for *this Program Invocation.*

Its representation in the *AddressSpace* is defined in Table 13.

Table 13 – ProgramDiagnostic2DataType definition

Attributes	Value
BrowseName	ProgramDiagnostic2DataType

5.2.9 ProgramDiagnostic2Type VariableType

This *VariableType* aggregates simple *Variables* using simple *DataTypes* that reflect the elements of the ProgramDiagnosticDataType structure. Its *DataVariables* have the same semantic as defined in in 5.2.8.

NOTE The original ProgramDiagnosticType VariableType had the same flaws as the structure. To avoid collisions with existing implementations, a new version with name ProgramDiagnostic2Type has been created.

The *VariableType* is formally defined in Table 14.

Table 14 – ProgramDiagnostic2Type VariableType

Attribute	Value			
BrowseName	ProgramDiagnostic2Type			
IsAbstract	False			
References	**NodeClass**	**BrowseName**	**DataType / TypeDefinition**	**Modelling Rule**
Subtype of the BaseDataVariableType defined in IEC 62541-5.				
HasComponent	Variable	CreateSessionId	NodeId	Mandatory
HasComponent	Variable	CreateClientName	String	Mandatory
HasComponent	Variable	InvocationCreationTime	UtcTime	Mandatory
HasComponent	Variable	LastTransitionTime	UtcTime	Mandatory
HasComponent	Variable	LastMethodCall	String	Mandatory
HasComponent	Variable	LastMethodSessionId	NodeId	Mandatory
HasComponent	Variable	LastMethodInputArguments	Argument[]	Mandatory
HasComponent	Variable	LastMethodOutputArguments	Argument[]	Mandatory
HasComponent	Variable	LastMethodInputValues	BaseDataType[]	Mandatory
HasComponent	Variable	LastMethodOutputValues	BaseDataType[]	Mandatory
HasComponent	Variable	LastMethodCallTime	UtcTime	Mandatory
HasComponent	Variable	LastMethodReturnStatus	StatusCode	Mandatory

Annex A
(informative)

Program example

A.1 Overview

This example illustrates the use of a *Program* to manage a domain download into a control system as depicted in Figure A.1. The download requires the segmented transfer of control operation data from a secondary storage device to the local memory within a control system.

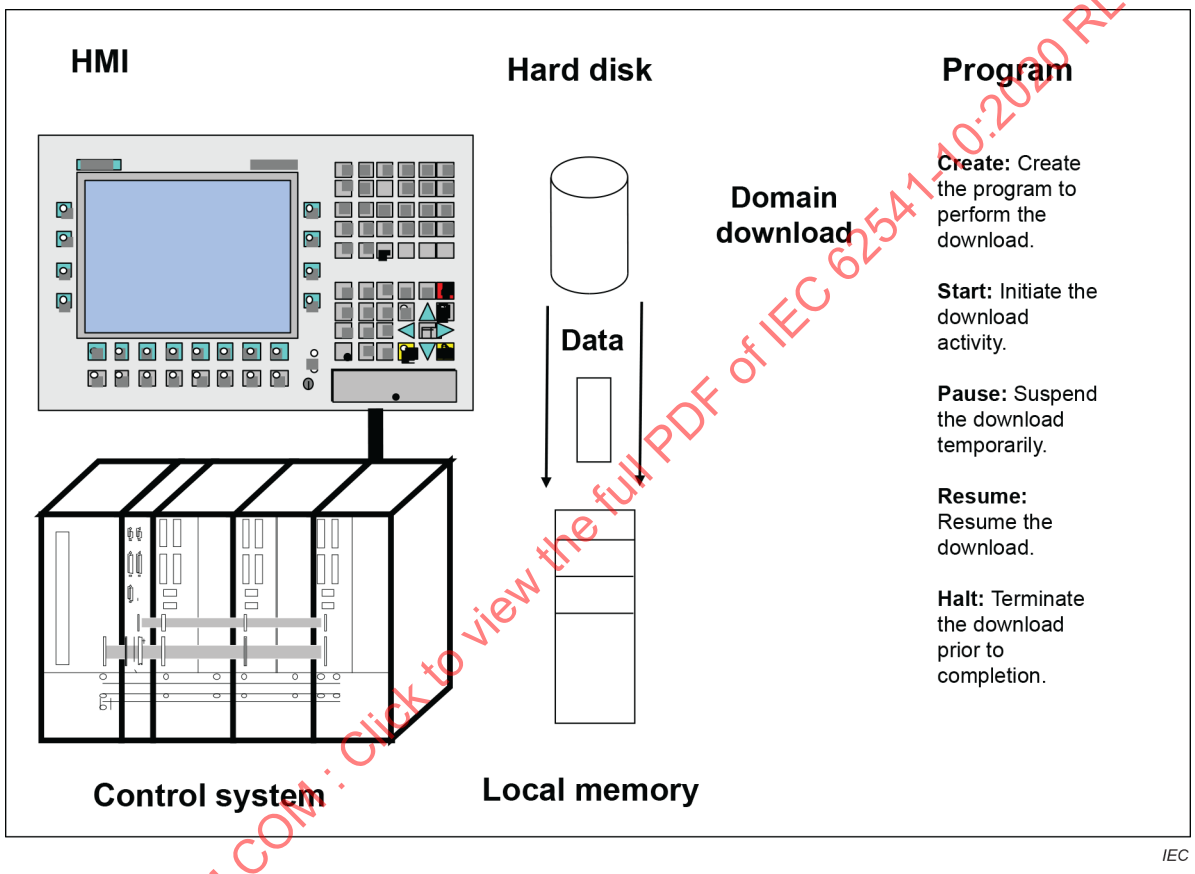


Figure A.1 – Program example

The domain download has a source and a target location which are identified when the download is initiated. Each time a segment of the domain is successfully transferred the *Client* is notified and informed of the amount of data that has been downloaded. The *Client* is also notified when the download is finished. The percentage of the total data received is reported periodically while the download continues. If the download fails, the cause of the failure is reported. At the completion of the download, the performance information is kept at the *Server*.

A.2 DomainDownload Program

A.2.1 General

The *Client* uses the "DomainDownload" *Program* to manage and monitor the download of a domain at the *Server*.

A.2.2 DomainDownload states

The basic state model for the DomainDownload *Program* is presented in Figure A.2. The *Program* has three primary states, Ready, Running, and Halted which are aligned with the standard states of a *ProgramStateMachineType*. Additionally, the *DomainDownloadType* extends the *ProgramStateMachineType* by defining subordinate *State Machines* for the *Program's* Running and Halted states. The subordinate states describe the download operations in greater detail and allow the *Client* to monitor the activity of the download at a finer resolution.

An instance (*Program Invocation*) of a DomainDownload *Program* is created by the *Client* each time a download is to be performed. The instance exists until explicitly removed by the *Client*. The initial state of the *Program* is Ready and the terminal state is Halted. The DomainDownload can be temporarily suspended and then resumed or aborted. Once halted, the program may not be restarted.

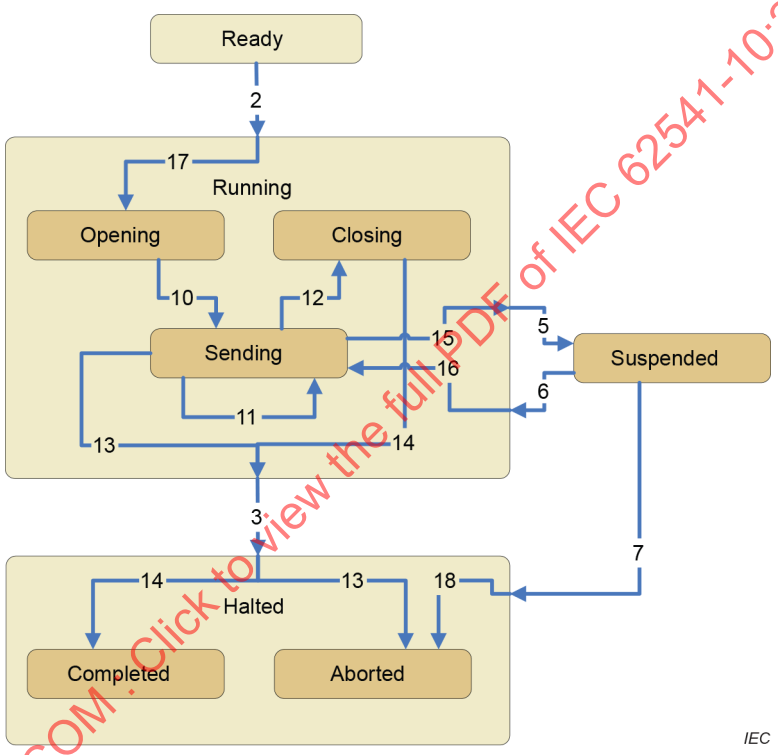


Figure A.2 – DomainDownload state diagram

The sequence of state transitions is illustrated in Figure A.2. Once the download is started, The *Program* progresses to the Opening state. After the source of the data is opened, a sequence of transfers occurs in the Sending state. When the transfer completes the *Objects* are closed in the Closing state. If the transfer is terminated before all of the data is downloaded or an error is encountered then the download is halted and the *Program* transitions to the Aborted state; otherwise the *Program* halts in the Completed state. The states are presented in Table A.1 along with the state transitions.

A.2.3 DomainDownload transitions

The valid state transitions specified for the DomainDownload *Program* are specified in Table A.1. Each of the transitions defines a start state and end state for the transition and is identified by a unique number. Five of the transitions are from the base *ProgramStateMachineType* and retain the transition identifier numbers specified for *Programs*. The additional transitions relate the base *Program* states with the subordinate states defined for the DomainDownload. These states have been assigned unique transition identifier numbers that distinguish them from the base *Program* transition identifiers. In cases

where transitions occur between substates and the *Program's* base states, two transitions are specified. One transition identifies the base state change and a second substate change. For example, ReadytoRunning and ToOpening occur at the same time.

Table A.1 also specifies the defined states, causes for the transitions, and the effects of each transition. *Program Control Methods* are used by the *Client* to "run" the DomainDownload. The *Methods* cause or trigger the specified transitions. The transition effects are the specified *EventTypes* which notify the *Client* of *Program* activity.

Table A.1 – DomainDownload states

No.	Transition name	Cause	From State	To State	Effect
2	ReadyToRunning	Start Method	Ready	Running	Report Transition 2 Event/Result
3	RunningToHalted	Halt Method/Error or Internal.	Running	Halted	Report Transition 3 Event/Result
5	RunningToSuspended	Suspend Method	Running	Suspended	Report Transition 5 Event/Result
6	SuspendedToRunning	Resume Method	Suspended	Running	Report Transition 6 Event/Result
7	SuspendedToHalted	Halt Method	Suspended	Halted	Report Transition 7 Event/Result
10	OpeningToSending	Internal	Opening	Sending	Report Transition 10 Event/Result
11	SendingToSending	Internal	Sending	Sending	Report Transition 11 Event/Result
12	SendingToClosing	Internal	Sending	Closing	Report Transition 12 Event/Result
13	SendingToAborted	Halt Method/Error	Opening	Aborted	Report Transition 13 Event/Result
14	ClosingToCompleted	Internal	Closing	Completed	Report Transition 14 Event/Result
15	SendingToSuspended	Suspend Method	Sending	Suspended	Report Transition 16 Event/Result
16	SuspendedToSending	Resume Method	Suspended	Sending	Report Transition 17 Event/Result
18	SuspendedToAborted	Halt Method	Suspended	Aborted	Report Transition 18 Event/Result
17	ToOpening	Internal	Ready	Opening	Report Transition 19 Event/Result

A.2.4 DomainDownload Methods

A.2.4.1 General

Four standard *Program Methods* are specified for running the DomainDownload *Program, Start, Suspend, Resume, and Halt.* No additional *Methods* are specified. The base behaviours of these *Methods* are defined by the *ProgramStateMachineType.* The *Start Method* initiates the download activity and passes the source and destination locations for the transfer. The *Suspend Method* is used to pause the activity temporarily. The *Resume Method* reinitiates the download, when paused. The *Halt Method* aborts the download. Each of the *Methods* causes a *Program* state transition and a substate transition. The specific state transition depends on the current state at the time the *Method* is called. If a *Method Call* is made when the DomainDownload is in a state for which that *Method* has no associated transition, the *Method* returns an error status indicating invalid state for the *Method*.

A.2.4.2 Method Arguments

The *Start Method* specifies three input arguments to be passed when it is called: Domain Name, DomainSource, and DomainDestination. The other *Methods* require no input arguments. No output arguments are specified for the DomainDownload *Methods*. The resultant error status for the *Program* is part of the *Call Service*.

A.2.5 DomainDownload Events

A.2.5.1 General

A *ProgramTransitionEventType* is specified for each of the DomainDownload *Program* transitions. The *EventTypes* trigger a specific *Event* notification to the *Client* when the associated state transition occurs in the running *Program* Instance. The *Event* notification identifies the transition. The SendingToSending state transition also includes intermediate result data.

A.2.5.2 Event information

The SendingToSending *Program* transition *Event* relays intermediate result data to the *Client* along with the notification. Each time the transition occurs, data items describing the amount and percentage of data transferred are sent to the *Client*.

A.2.5.3 Final result data

The DomainDownload *Program* retains final result data following a completed or aborted download. The data includes the total transaction time and the size of the domain. In the event of an aborted download, the reason for the termination is retained.

A.2.6 DomainDownload model

A.2.6.1 Overview

The OPC UA model for the DomainDownload *Program* is presented in A.2.6.2. Collectively they define the components that constitute this *Program*. For clarity, the figures present a progression of portions of the model that complement the contents of the tables and illustrate the *Program's* composition.

The type definition for the DomainDownload *Program* precisely represents the behaviour of the *Program* in terms of OPC UA components. These components can be browsed by a *Client* to interpret or validate the actions of the *Program*.

A.2.6.2 DomainDownloadType

The DomainDownloadType is a subtype derived from the *ProgramStateMachineType*. It specifies the use or non-use of optional *ProgramStateMachineType* components, valid extensions such as subordinate *State Machines*, and constrained attribute values applied to instances of DomainDownload *Programs*.

Table A.2 specifies the optional and extended components defined by the DomainDownload Type. Note the references to two sub *State Machine Types*, *TransferStateMachine* and *FinishStateMachine*. The DomainDownloadType omits references to the *Reset Program Control Method* and its associated state transition (HaltedToReady), which it does not support.

Table A.2 – DomainDownload Type

Attribute	Value				
	Includes all non-optional attributes specified for the *ProgramStateMachineType*				
BrowseName	DomainDownloadType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**Data Type**	**TypeDefinition**	**Modelling Rule**
HasComponent	Object	TransferStateMachine		StateMachineType	Mandatory
HasComponent	Object	FinishStateMachine		StateMachineType	Mandatory
HasComponent	Variable	ProgramDiagnostic		ProgramDiagnostic2Type	Mandatory
HasComponent	Object	ReadyToRunning		TransitionType	–
HasComponent	Object	RunningToHalted		TransitionType	–
HasComponent	Object	RunningToSuspended		TransitionType	–
HasComponent	Object	SuspendedToRunning		TransitionType	–
HasComponent	Object	SuspendedToHalted		TransitionType	–
HasComponent	Method	Start			Mandatory
HasComponent	Method	Suspend			Mandatory
HasComponent	Method	Halt			Mandatory
HasComponent	Method	Resume			Mandatory
HasComponent	Object	FinalResultData		BaseObjectType	Mandatory

Table A.3 specifies the *Transfer State Machine type* that is a sub *State Machine* of the DomainDownload *Program Type*. This definition identifies the *StateTypes* that compose the substates for the *Program's* Running *StateType*.

Table A.3 – Transfer State Machine Type

Attribute	Value				
	Includes all attributes specified for the FiniteStateMachineType				
BrowseName	TransferStateMachineType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**Data Type**	**TypeDefinition**	**Modelling Rule**
HasComponent	Object	Opening		StateType	–
HasComponent	Object	Sending		StateType	–
HasComponent	Object	Closing		StateType	–
HasComponent	Object	ReadyToOpening		TransitionType	–
HasComponent	Object	OpeningToSending		TransitionType	–
HasComponent	Object	SendingToClosing		TransitionType	–
HasComponent	Object	SendingToAborted		TransitionType	–
HasComponent	Object	SendingToSuspended		TransitionType	–
HasComponent	Object	SuspendedToSending		TransitionType	–
HasComponent	Method	Start			Mandatory
HasComponent	Method	Suspend			Mandatory
HasComponent	Method	Halt			Mandatory
HasComponent	Method	Resume			Mandatory

Table A.3 specifies the *StateTypes* associated with the Transfer State Machine Type. All of these states are substates of the *Running* state of the base *ProgramStateMachineType*.

The Opening state is the preparation state for the domain download.

The Sending state is the activity state for the transfer in which the data is moved from the source to destination.

The Closing state is the cleanup phase of the download.

Table A.4 defines the states of the *TransferStateMachineType*.

Table A.4 – Transfer State Machine – states

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	NOTES
States					
Opening	HasProperty	StateNumber	5	PropertyType	
	ToTransition	OpeningToSending		TransitionType	
	FromTransition	ToOpening		TransitionType	
	ToTransition	OpeningToSending		TransitionType	
Sending	HasProperty	StateNumber	6	PropertyType	
	FromTransition	OpeningToSending		TransitionType	
	ToTransition	SendingToSending		TransitionType	
	ToTransition	SendingToClosing		TransitionType	
	ToTransition	SendingToSuspended		TransitionType	
	FromTransition	ToSending		TransitionType	
Closing	HasProperty	StateNumber	7	PropertyType	
	ToTransition	ClosingToCompleted		TransitionType	
	ToTransition	ClosingToAborted		TransitionType	
	FromTransition	SendingToClosing		TransitionType	

Table A.5 specifies the *Finish State Machine Type* that is a substate machine of the DomainDownload *ProgramStateMachineType*. This definition identifies the *StateTypes* that compose the substates for the *Program's* Halted *StateType*.

Table A.5 – Finish State Machine Type

Attribute	Value				
	Includes all attributes specified for the FiniteStateMachineType				
BrowseName	TransferStateMachineType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**Data Type**	**TypeDefinition**	**Modelling Rule**
HasComponent	Object	Completed		StateType	–
HasComponent	Object	Aborted		StateType	–

Table A.6 specifies the *StateTypes* associated with the *Finish State Machine Type*. Note these are final states and that they have no associated transitions between them.

Table A.6 – Finish State Machine – states

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	NOTES
States					
Aborted	HasProperty	StateNumber	8	PropertyType	
	FromTransition	OpeningToAborted		TransitionType	
	FromTransition	ClosingToAborted		TransitionType	
Completed	HasProperty	StateNumber	9	PropertyType	
	FromTransition	ClosingToCompleted		TransitionType	

The Aborted state is the terminal state that indicates an incomplete or failed domain download operation.

The Completed state is the terminal state that indicates a successful domain download.

Table A.7 specifies the constraining behaviour of a DomainDownload.

Table A.7 – DomainDownload Type Property Attributes variable values

NodeClass	BrowseName	Data Type	Data Value	Modelling Rule
Variable	Creatable	Boolean	True	–
Variable	Deletable	Boolean	True	Mandatory
Variable	AutoDelete	Boolean	False	Mandatory
Variable	RecycleCount	Int32	0	Mandatory
Variable	InstanceCount	UInt32	PropertyType	–
Variable	MaxInstanceCount	UInt32	500	–
Variable	MaxRecycleCount	UInt32	0	–

A DomainDownload *Program Invocation* can be created and also destroyed by a *Client*. The *Program Invocation* will not delete itself when halted, but will persist until explicitly removed by the *Client*. A DomainDownload *Program Invocation* cannot be reset to restart. The *Server* will support up to 500 concurrent DomainDownload *Program Invocations*.

Figure A.3 presents a partial DomainDownloadType model that illustrates the association between the states and the DomainDownload, Transfer, and Finish State Machines. Note that the current state number for the substate machines is only valid when the DomainDownload active base state references the substate machine, Running for the Transfer current state and Halted for the Finish current state.

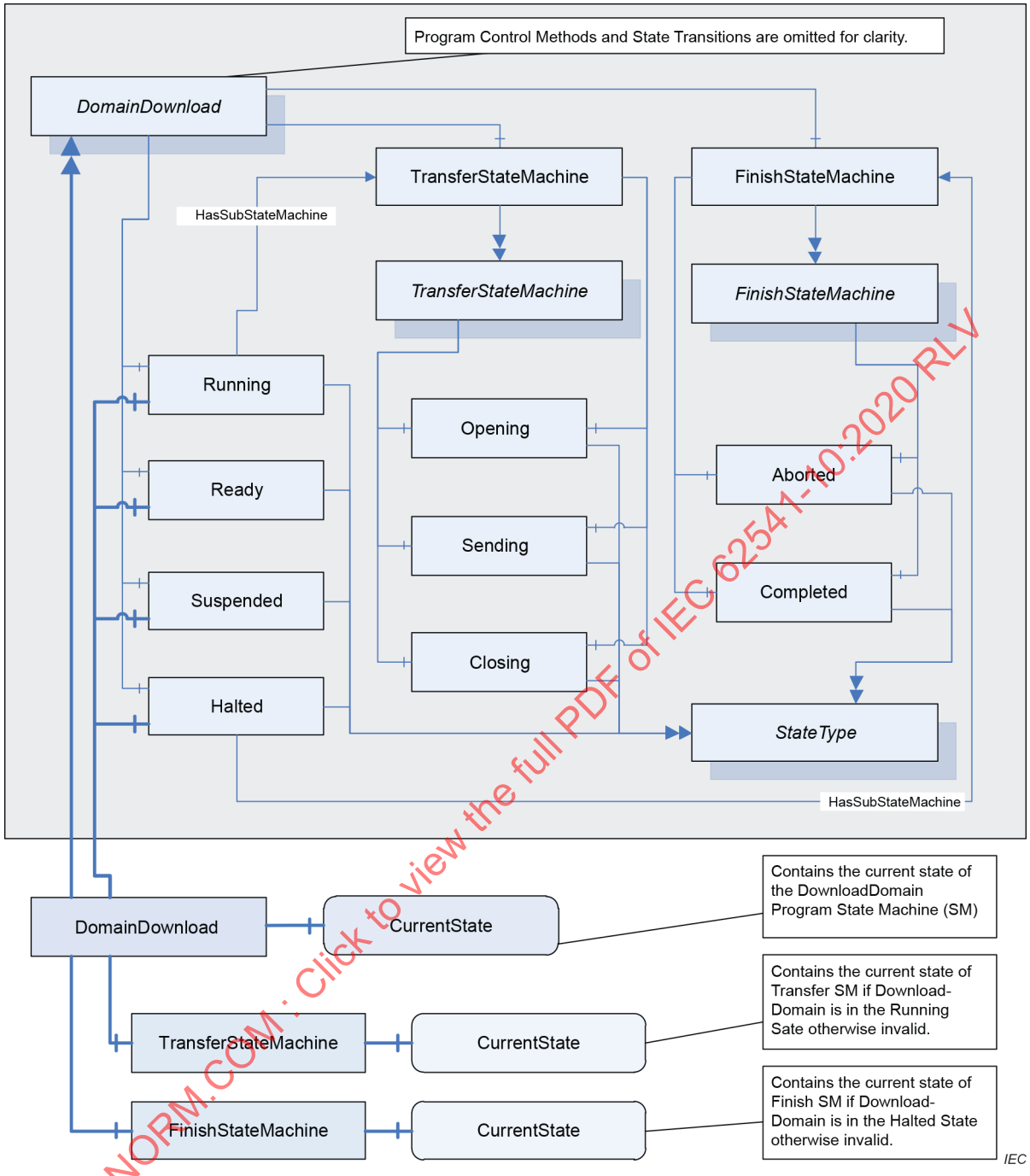


Figure A.3 – DomainDownloadType partial state model

Table A.8 specifies the *ProgramTransitionTypes* that are defined in addition to the *ProgramTransitionTypes* specified for *Programs* in Table 7. These types associate the Transfer and Finish substate machine states with the states of the base *Program*.

Table A.8 – Additional DomainDownload transition types

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	NOTES
Transitions					
ToSending	HasProperty	TransitionNumber	10	PropertyType	
	ToState	Sending		StateType	
	FromState	Opening		StateType	
	HasCause	Start			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
SendingToSending	HasProperty	TransitionNumber	11	PropertyType	
	ToState	Sending		StateType	
	FromState	Sending		StateType	
	HasEffect	ProgramTransitionEventType			
SendingToClosing	HasProperty	TransitionNumber	12	PropertyType	
	ToState	Closing		StateType	
	FromState	Sending		StateType	
	HasEffect	ProgramTransitionEventType			
SendingToAborted	HasProperty	TransitionNumber	13	PropertyType	
	ToState	Aborted		StateType	
	FromState	Sending		StateType	
	HasCause	Halt			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
ClosingToCompleted	HasProperty	TransitionNumber	14	PropertyType	
	ToState	Completed		StateType	
	FromState	Closing		StateType	
	HasEffect	ProgramTransitionEventType			
SendingToSuspended	HasProperty	TransitionNumber	15	PropertyType	
	ToState	Suspended		StateType	
	FromState	Sending		StateType	
	HasCause	Suspend			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			

BrowseName	References	Target BrowseName	Value	Target TypeDefinition	NOTES
Transitions					
SuspendedToSending	HasProperty	TransitionNumber	16	PropertyType	
	ToState	Sending		StateType	
	FromState	Suspended		StateType	
	HasCause	Resume			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
SuspendedToAborted	HasProperty	TransitionNumber	18	PropertyType	
	ToState	Aborted		StateType	
	FromState	Suspended		StateType	
	HasCause	Halt			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
ReadyToOpening	HasProperty	TransitionNumber	17	PropertyType	
	ToState	Opening		StateType	
	FromState	Ready		StateType	
	HasCause	Start			Method
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			

Figure A.4 through Figure A.10 illustrate portions of the DomainDownloadType model. In each figure, the referenced tates, *Methods*, transitions, and *EventTypes* are identified for one or two state transitions.

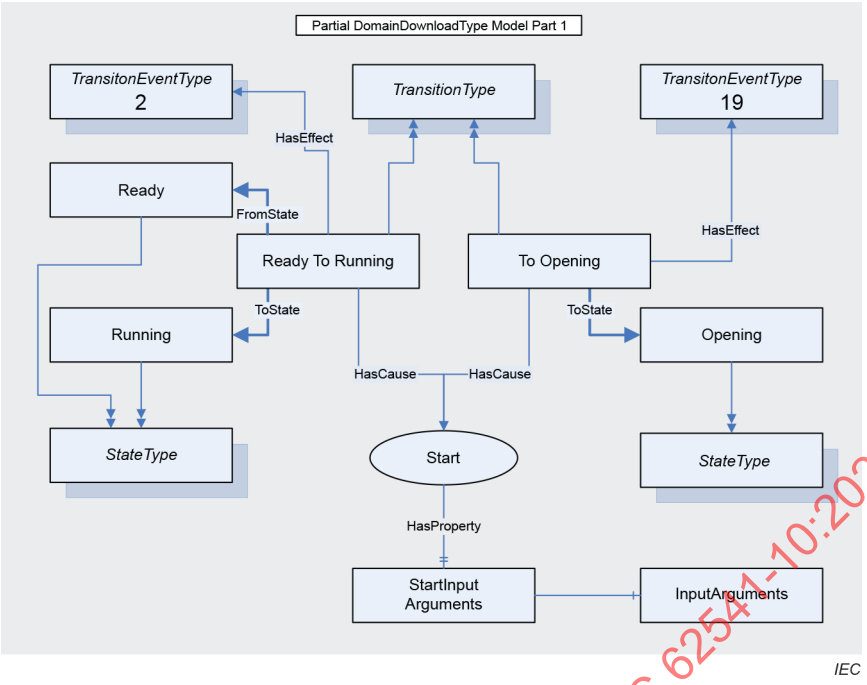


Figure A.4 – Ready To Running model

Figure A.4 illustrates the model for the ReadyToRunning *Program* transition. The transition is caused by the *Start Method*. The *Start Method* requires three input arguments. The *Method Call* service is used by the *Client* to invoke the *Start Method* and pass the arguments. When successful, the *Program Invocation* enters the Running state and the subordinate Transfer Opening state. The *Server* issues two *Event* notifications, ReadyToRunning (2), and ToOpening (19).

Table A.9 – Start Method additions

Attribute	**Value**				
BrowseName	Start				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArgument	Argument[]	PropertyType	–

Table A.9 specifies that the *Start Method* for the *DomainDownloadType* requires input arguments. Table A.10 identifies the *Start Arguments* required.

Table A.10 – StartArguments

Name	Type	Value
Argument 1	structure	
name	String	SourcePath
dataType	NodeId	StringNodeId
valueRank	Int32	-1 (-1 = scalar)
arrayDimensions	UInt32[]	null
description	LocalizedText	The source specifier for the domain
Argument 2	structure	
Name	String	DestinationPath
dataType	NodeId	StringNodeId
valueRank	Int32	-1 (-1 = scalar)
arrayDimensions	UInt32[]	null
description	LocalizedText	The destination specifier for the domain
Argument 3	structure	
name	String	DomainName
dataType	NodeId	StringNodeId
arrayDimensions	UInt32[]	null
valueRank	Int32	-1 (-1 = scalar)
description	LocalizedText	The name of the domain

Figure A.5 illustrates the model for the Opening To Sending and the Sending to Closing *Program* transitions. As specified in the transition table, these state transitions require no *Methods* to occur, but rather are driven by the internal actions of the *Server*. *Events* are generated for each state transition (10 to 12), when they occur.

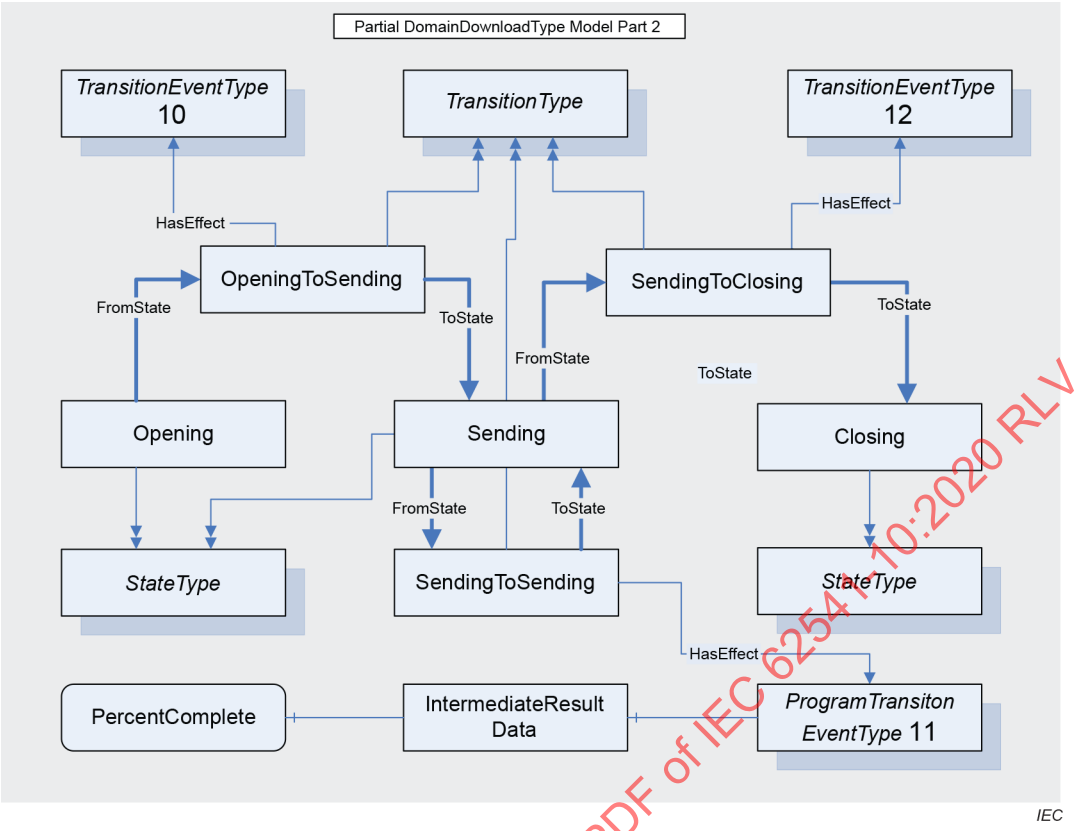


Figure A.5 – Opening To Sending To Closing model

Notice that a state transition can initiate and terminate at the same state (Sending). In this case the transition serves a purpose. The *ProgramTransitionEventType* effect referenced by the SendingToSending state transition has an *IntermediateResultData Object Reference*. The *IntermediateResultData Object* serves to identify two *Variables* whose values are obtained each time the state transition occurs. The values are sent to the *Client* with the *Event* notification. Table A.11 defines the *IntermediateResults ObjectType* and Table A.12 defines the *Variables* of the *ObjectType*.

Table A.11 – IntermediateResults Object

Attribute	Value				
	Includes all attributes specified for the ObjectType				
BrowseName	IntermediateResults				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**Data Type**	**TypeDefinition**	**Modelling Rule**
HasComponent	Variable	AmountTransferred	Long	VariableType	Mandatory
HasComponent	Variable	PercentageTransferred	Long	VariableType	Mandatory

Table A.12 – Intermediate result data Variables

Intermediate Result Variables	Type	Value
Variable 1	Structure	
Name	String	AmountTransferred
dataType	NodeId	StringNodeId
description	LocalizedText	Bytes of domain data transferred.
Variable 2	Structure	
Name	String	PercentageTransferred
dataType	NodeId	StringNodeId
description	LocalizedText	Percentage of domain data transferred.

The model for the Running To Suspended state transition is illustrated in Figure A.6. The cause for this transition is the *Suspend Method.* The *Client* can pause the download of domain data to the control. The transition from Running to Suspended invokes the *Event* generation for *TransitionEventTypes* 5 and 16. Note that there is no longer a valid current state for the Transfer State Machine.

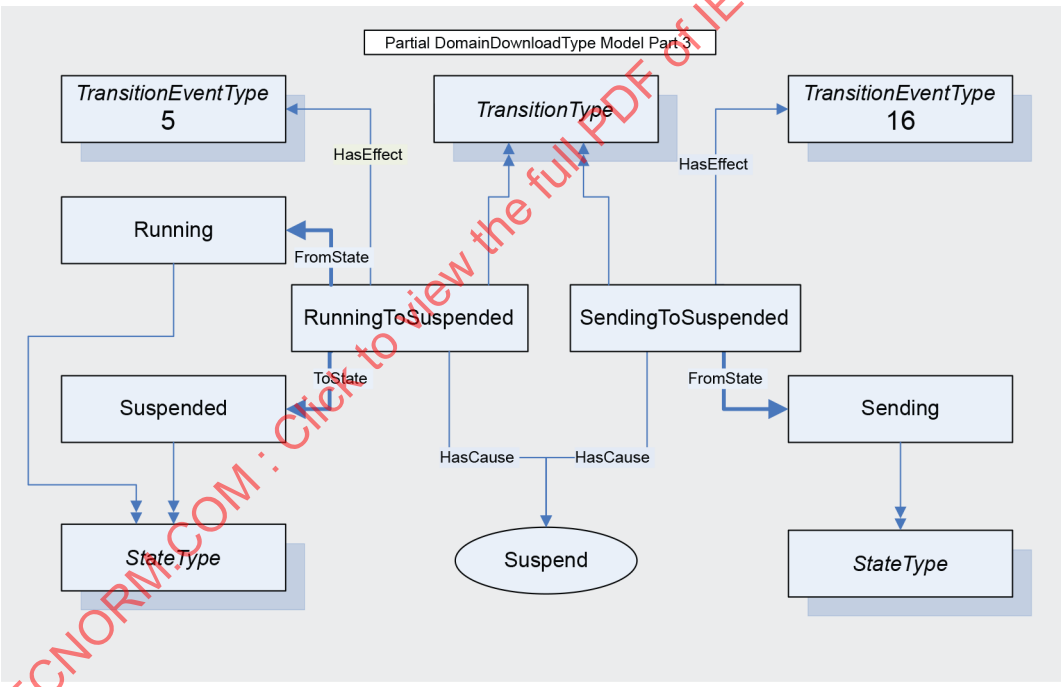


Figure A.6 – Running To Suspended model

The model for the SuspendedToRunning state transition is illustrated in Figure A.7. The cause for this transition is the *Resume Method.* The *Client* can resume the download of domain data to the control. The transition from Suspended to Running generates the *Event* for *TransitionEventTypes* 6 and 17. Now that the Running state is active, the Sending state of the Transfer State Machine is again specified for the *CurrentStateNumber*.

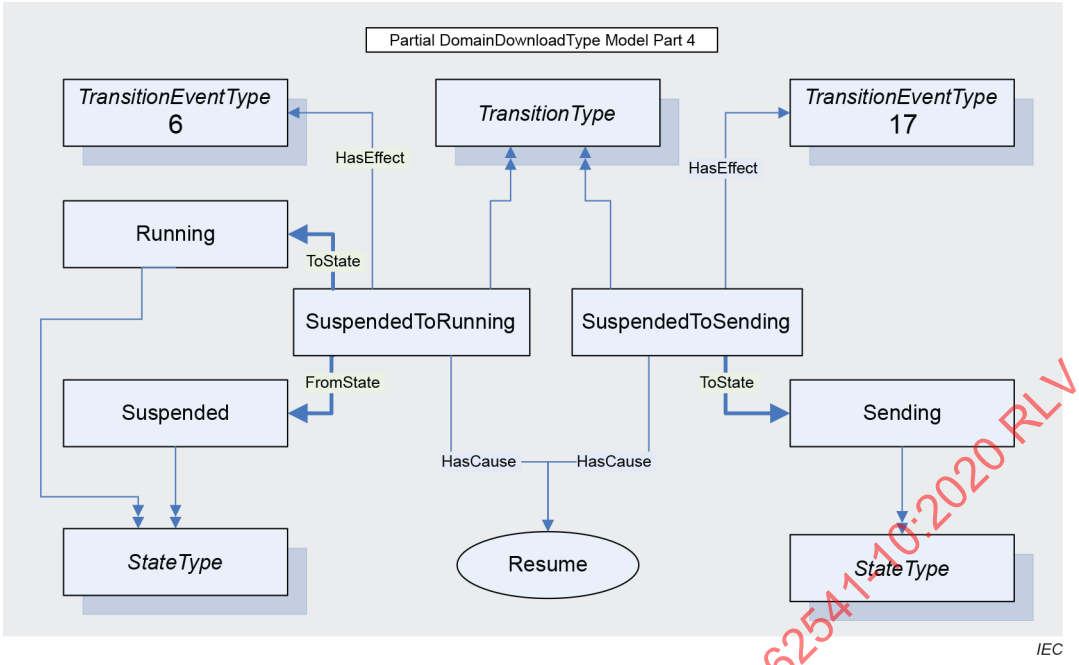


Figure A.7 – Suspended To Running model

The model for the Running To Halted state transition for an abnormal termination of the domain download is illustrated in Figure A.8. The cause for this transition is the *Halt Method.* The *Client* can terminate the download of domain data to the control. The transition from Running To Halted generates the *Event* for *TransitionEventTypes* 3 and 15. The *TransitionEventType* 15 indicates the transition from the Sending state as the Running State ends and then to the Aborted state as the Halted state is entered.

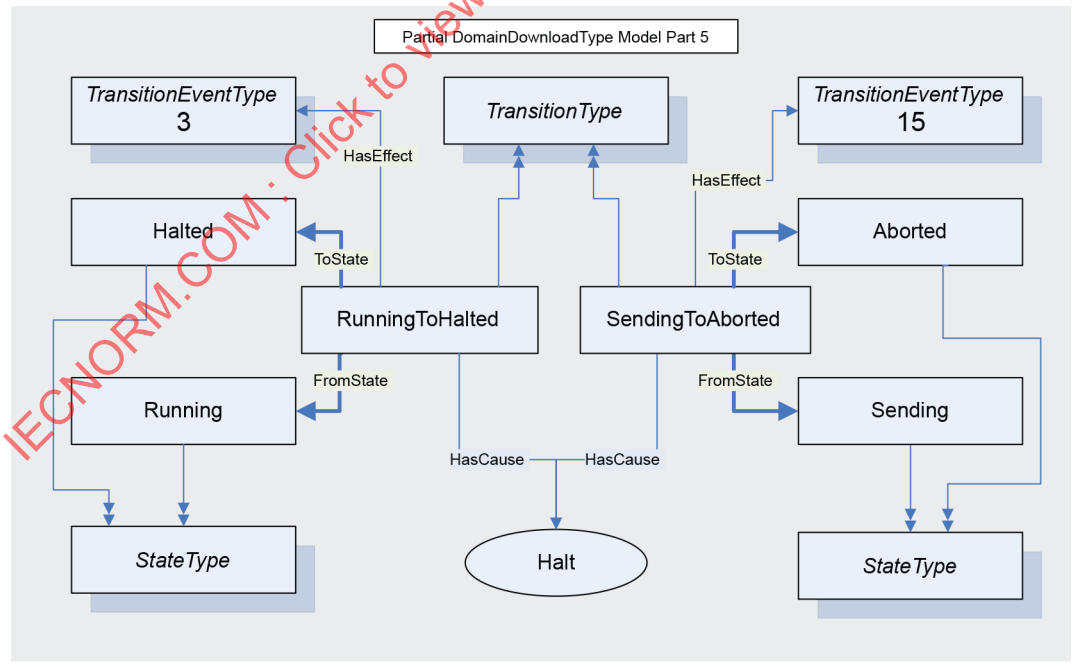


Figure A.8 – Running To Halted – Aborted model

Figure A.9 illustrates the model for the Suspended To Halted state transition for an abnormal termination of the domain download. The cause for this transition is the *Halt Method.* The *Client* can terminate the download of domain data to the control while it is suspended. The

transition from SuspendedToHalted invokes the *Event* notifiers for *TransitionEventTypes* 7 and 18.

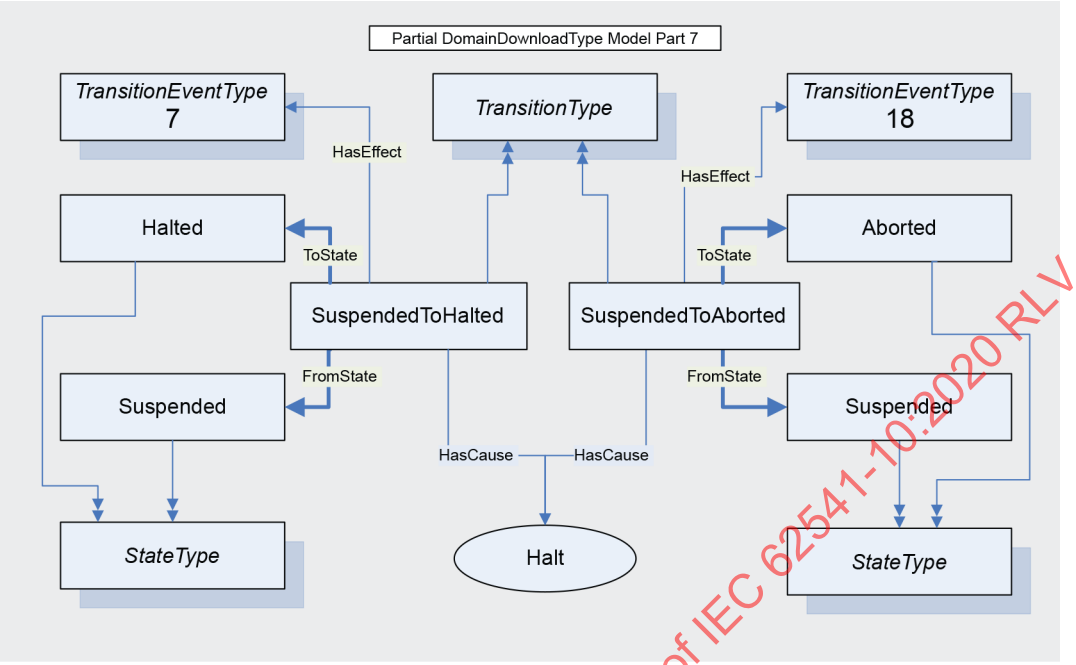


Figure A.9 – Suspended To Aborted model

The model for the Running To Completed state transition for a normal termination of the domain download is illustrated in Figure A.10. The cause for this transition is internal. The transition from Closing To Halted generates the Event for *TransitionEventTypes* 3 and 14. The *TransitionEventType* 14 indicates the transition from the Closing state as the Running state ends and then to the Completed state as the Halted state is entered.

The DomainDownloadType includes a component reference to a *FInalResultData Object*. This *Object* references *Variables* that persists information about the domain download once it has completed. This data can be read by *Clients* who are not subscribed to *Event* notifications. The result data is described in Table A.13.

Table A.13 – FinalResultData

Attribute	Value				
	Includes all attributes specified for the ObjectType				
BrowseName	FinalResultData				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**Data Type**	**TypeDefinition**	**Modelling Rule**
HasComponent	Variable	DownloadPerformance	Double	VariableType	Mandatory
HasComponent	Variable	FailureDetails	String	VariableType	Mandatory

The Domain Download net transfer data rate and detailed reason for aborted downloads are retained as final result data for each *Program Invocation*.

DownloadPerformance provides the data rate in seconds for domain data transferred.

FailureDetails provides a descriptive reason for an abort.

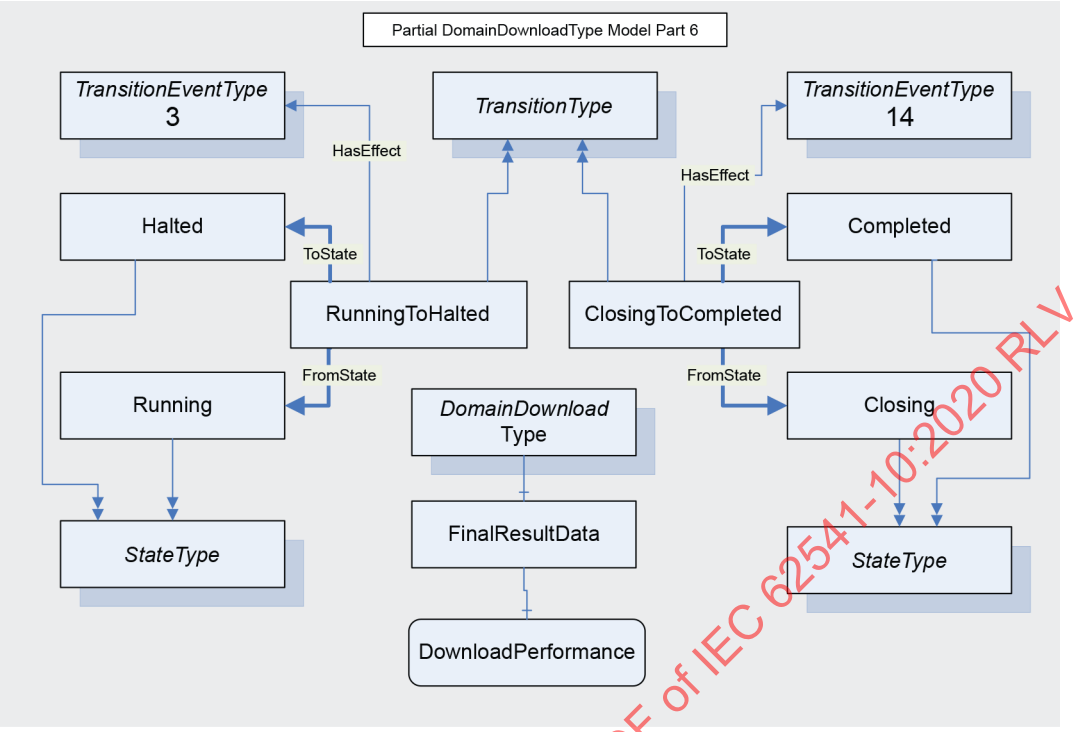


Figure A.10 – Running To Completed model

A.2.6.3 Sequence of operations

Figure A.11 illustrates a normal sequence of service exchanges between a *Client* and *Server* that would occur during the life cycle of a DomainDownloadType *Program Invocation*.

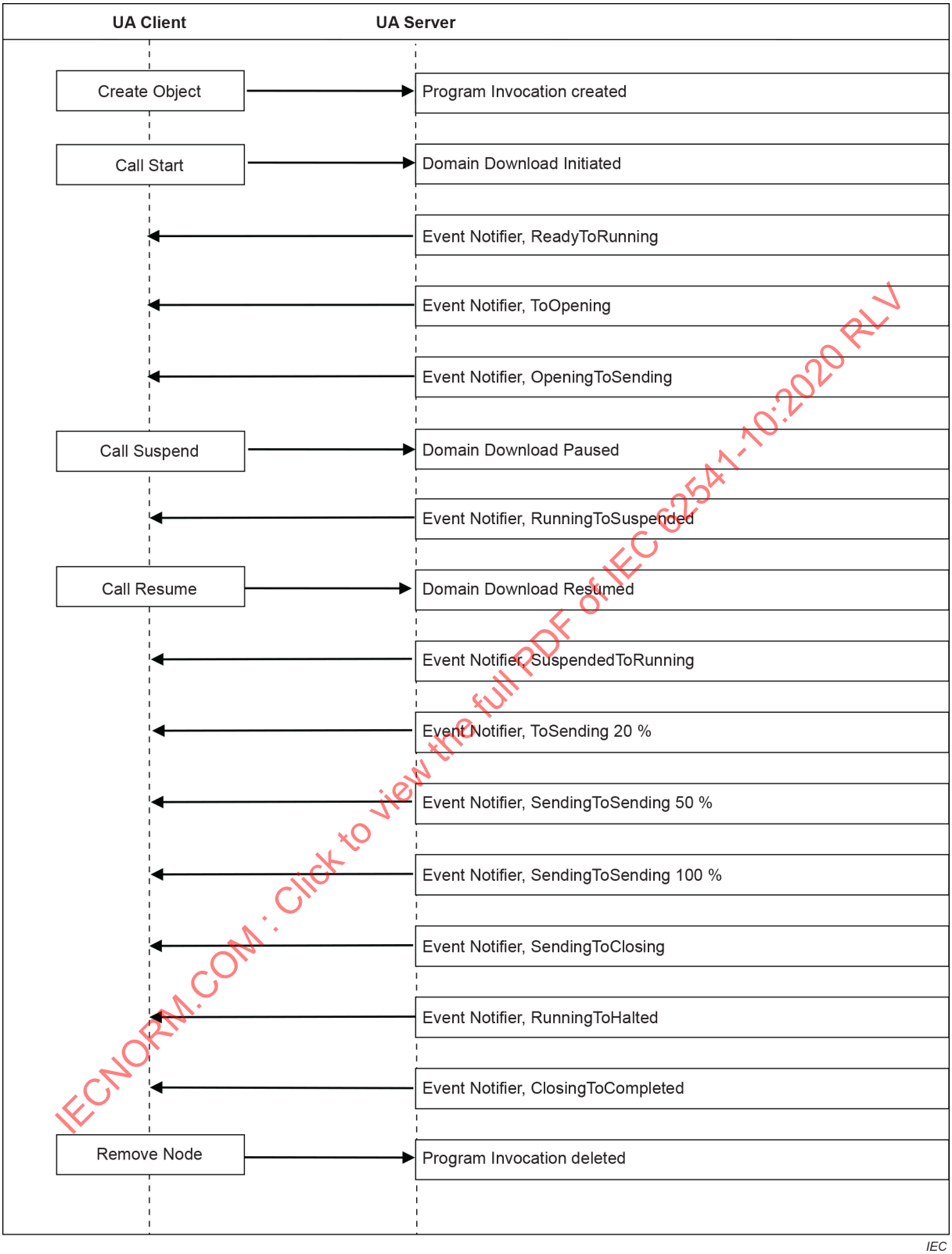


Figure A.11 – Sequence of operations

SOMMAIRE

COMMISSION ÉLECTROTECHNIQUE INTERNATIONALE

_____

ARCHITECTURE UNIFIÉE OPC –

Partie 10: Programmes

AVANT-PROPOS

1) La Commission Electrotechnique Internationale (IEC) est une organisation mondiale de normalisation composée de l'ensemble des comités électrotechniques nationaux (Comités nationaux de l'IEC). L'IEC a pour objet de favoriser la coopération internationale pour toutes les questions de normalisation dans les domaines de l'électricité et de l'électronique. A cet effet, l'IEC – entre autres activités – publie des Normes internationales, des Spécifications techniques, des Rapports techniques, des Spécifications accessibles au public (PAS) et des Guides (ci-après dénommés "Publication(s) de l'IEC"). Leur élaboration est confiée à des comités d'études, aux travaux desquels tout Comité national intéressé par le sujet traité peut participer. Les organisations internationales, gouvernementales et non gouvernementales, en liaison avec l'IEC, participent également aux travaux. L'IEC collabore étroitement avec l'Organisation Internationale de Normalisation (ISO), selon des conditions fixées par accord entre les deux organisations.
2) Les décisions ou accords officiels de l'IEC concernant les questions techniques représentent, dans la mesure du possible, un accord international sur les sujets étudiés, étant donné que les Comités nationaux de l'IEC intéressés sont représentés dans chaque comité d'études.
3) Les Publications de l'IEC se présentent sous la forme de recommandations internationales et sont agréées comme telles par les Comités nationaux de l'IEC. Tous les efforts raisonnables sont entrepris afin que l'IEC s'assure de l'exactitude du contenu technique de ses publications; l'IEC ne peut pas être tenue responsable de l'éventuelle mauvaise utilisation ou interprétation qui en est faite par un quelconque utilisateur final.
4) Dans le but d'encourager l'uniformité internationale, les Comités nationaux de l'IEC s'engagent, dans toute la mesure possible, à appliquer de façon transparente les Publications de l'IEC dans leurs publications nationales et régionales. Toutes divergences entre toutes Publications de l'IEC et toutes publications nationales ou régionales correspondantes doivent être indiquées en termes clairs dans ces dernières.
5) L'IEC elle-même ne fournit aucune attestation de conformité. Des organismes de certification indépendants fournissent des services d'évaluation de conformité et, dans certains secteurs, accèdent aux marques de conformité de l'IEC. L'IEC n'est responsable d'aucun des services effectués par les organismes de certification indépendants.
6) Tous les utilisateurs doivent s'assurer qu'ils sont en possession de la dernière édition de cette publication.
7) Aucune responsabilité ne doit être imputée à l'IEC, à ses administrateurs, employés, auxiliaires ou mandataires, y compris ses experts particuliers et les membres de ses comités d'études et des Comités nationaux de l'IEC, pour tout préjudice causé en cas de dommages corporels et matériels, ou de tout autre dommage de quelque nature que ce soit, directe ou indirecte, ou pour supporter les coûts (y compris les frais de justice) et les dépenses découlant de la publication ou de l'utilisation de cette Publication de l'IEC ou de toute autre Publication de l'IEC, ou au crédit qui lui est accordé.
8) L'attention est attirée sur les références normatives citées dans cette publication. L'utilisation de publications référencées est obligatoire pour une application correcte de la présente publication.
9) L'attention est attirée sur le fait que certains des éléments de la présente Publication de l'IEC peuvent faire l'objet de droits de brevet. L'IEC ne saurait être tenue pour responsable de ne pas avoir identifié de tels droits de brevets et de ne pas avoir signalé leur existence.

L'IEC 62541-10 a été établie par le sous-comité 65E: Les dispositifs et leur intégration dans les systèmes de l'entreprise, du comité d'études 65 de l'IEC: Mesure, commande et automation dans les processus industriels.

Cette troisième édition annule et remplace la deuxième édition parue en 2015.

Cette édition inclut plusieurs clarifications ainsi que les modifications techniques majeures suivantes par rapport à l'édition précédente:

a) remplacement de ProgramType par ProgramStateMachineType, conformément au NodeSet (et donc aux mises en œuvre). Dans le programme ProgramDiagnosticDataType, modification des définitions de lastInputArguments et lastOutputArguments, et ajout de deux champs complémentaires pour les valeurs d'arguments. Remplacement également de StatusResult par StatusCode. Création d'une nouvelle version du type de ProgramDiagnostic2DataType;

b) remplacement de la règle de modélisation Facultative par OptionalPlaceHolder pour les Méthodes de Commande de Programme. Ceci fait suite à la clarification apportée dans l'IEC 62541-3, et permet aux sous-types (ou instances) d'ajouter des arguments.

Le texte de cette norme est issu des documents suivants:

FDIS	Rapport de vote
65E/719/FDIS	65E/735/RVD

Le rapport de vote indiqué dans le tableau ci-dessus donne toute information sur le vote ayant abouti à l'approbation de cette Norme internationale.

Ce document a été rédigé selon les Directives ISO/IEC, Partie 2.

Dans l'ensemble du présent document et dans les autres parties de la série IEC 62541, certaines conventions de document sont utilisées:

Le format *italique* est utilisé pour mettre en évidence un terme défini ou une définition qui apparaît à l'Article 3 dans l'une des parties de la série.

Le format *italique* est également utilisé pour mettre en évidence le nom d'un paramètre d'entrée ou de sortie de service, ou le nom d'une structure ou d'un élément de structure habituellement défini dans les tableaux.

Par ailleurs, les *termes* et les *noms en italique* sont, à quelques exceptions près, écrits en camel-case (pratique qui consiste à joindre, sans espace, les éléments des mots ou expressions composés, la première lettre de chaque élément étant en majuscule). Par exemple, le terme défini est *AddressSpace* et non Espace d'adressage. Cela permet de mieux comprendre qu'il existe une définition unique pour *AddressSpace*, et non deux définitions distinctes pour Espace et pour Adressage.

Une liste de toutes les parties de la série IEC 62541, publiées sous le titre général *Architecture unifiée OPC*, peut être consultée sur le site web de l'IEC.

Le comité a décidé que le contenu de ce document ne sera pas modifié avant la date de stabilité indiquée sur le site web de l'IEC sous "http://webstore.iec.ch" dans les données relatives au document recherché. A cette date, le document sera

- reconduit,
- supprimé,
- remplacé par une édition révisée, ou
- amendé.

IMPORTANT – Le logo "colour inside" qui se trouve sur la page de couverture de cette publication indique qu'elle contient des couleurs qui sont considérées comme utiles à une bonne compréhension de son contenu. Les utilisateurs devraient, par conséquent, imprimer cette publication en utilisant une imprimante couleur.

ARCHITECTURE UNIFIÉE OPC –

Partie 10: Programmes

1 Domaine d'application

La présente partie de l'IEC 62541 définit le *modèle d'information* associé avec des *Programmes* dans l'Architecture unifiée OPC. Elle comprend la description des *NodeClasses*, des *Propriétés*, *Méthodes* et *Evénements* normalisés et du comportement associé ainsi que des informations relatives aux *Programmes*.

Le modèle d'espace d'adressage complet, comprenant toutes les *NodeClasses* et tous les *Attributs*, est spécifié dans l'IEC 62541-3. Les *Services* tels que ceux utilisés pour invoquer les *Méthodes* appliquées pour gérer les *Programmes* sont spécifiés dans l'IEC 62541-4.

2 Références normatives

Les documents suivants sont cités dans le texte de sorte qu'ils constituent, pour tout ou partie de leur contenu, des exigences du présent document. Pour les références datées, seule l'édition citée s'applique. Pour les références non datées, la dernière édition du document de référence s'applique (y compris les éventuels amendements).

IEC TR 62541-1, *OPC Unified Architecture – Part 1: Overview and Concepts* (disponible en anglais seulement)

IEC 62541-3, *Architecture unifiée OPC – Partie 3: Modèle d'espace d'adressage*

IEC 62541-4, *Architecture unifiée OPC – Partie 4: Services*

IEC 62541-5, *Architecture unifiée OPC – Partie 5: Modèle d'Information*

IEC 62541-7, *Architecture unifiée OPC – Partie 7: Profils*

3 Termes, définitions et termes abrégés

3.1 Termes et définitions

Pour les besoins du présent document, les termes et définitions de l'IEC TR 62541-1, l'IEC 62541-3 ainsi que les suivants, s'appliquent.

L'ISO et l'IEC tiennent à jour des bases de données terminologiques destinées à être utilisées en normalisation, consultables aux adresses suivantes:

- IEC Electropedia: disponible à l'adresse http://www.electropedia.org/
- ISO Online browsing platform: disponible à l'adresse http://www.iso.org/obp

3.1.1
fonction
tâche de programmation assurée par un *Serveur* ou un dispositif, généralement réalisée par l'exécution de code machine

3.1.2
diagramme d'états finis
séquence d'états et de transitions d'états valides, associés aux causes et effets de ces transitions, définissant les actions d'un *Programme* en ce qui concerne les stades discrets

3.1.3
ProgramStateMachineType
définition de type d'un *Programme*, qui constitue un sous-type du *FiniteStateMachineType*

3.1.4
méthode de commande de programme
Méthode à la sémantique particulière conçue pour commander un *Programme* en déclenchant une transition d'états

3.1.5
invocation de programme
instance d'*Objet* unique d'un *Programme* existant sur un *Serveur*

Note 1 à l'article: L'*Invocation de Programme* se différencie des autres instances d'*Objet* du même *ProgramStateMachineType* par le chemin de navigation unique du nœud d'objet.

3.2 Termes abrégés

DA data access (accès aux données)

FSM finite state machine (diagramme d'états finis)

IHM interface homme–machine

UA unified architecture (architecture unifiée)

4 Concepts

4.1 Généralités

Les installations d'automatisation intégrée gèrent leurs opérations par des échanges de données et par l'invocation coordonnée de *Fonctions* système, comme représenté à la Figure 1. Il est exigé des *Services* qu'ils réalisent les échanges de données et qu'ils invoquent les *Fonctions* nécessaires à l'exploitation du système. Ces *Fonctions* peuvent être invoquées au moyen des Interfaces Homme-Machine, des contrôleurs cellulaires ou d'autres systèmes de commande de surveillance et d'acquisition de données. L'OPC UA définit les *Méthodes* et les *Programmes* comme des outils d'interopérabilité permettant d'annoncer, de découvrir et d'interroger ces *Fonctions*. Ils fournissent un mécanisme de normalisation applicable à la description sémantique, à l'invocation et à la consignation des résultats de ces *Fonctions*. Les *Méthodes* et les *Programmes* viennent compléter les autres *Services* et *ObjectTypes* OPC UA afin de faciliter l'exploitation d'un environnement d'automatisation fondé sur une hiérarchie client-serveur.

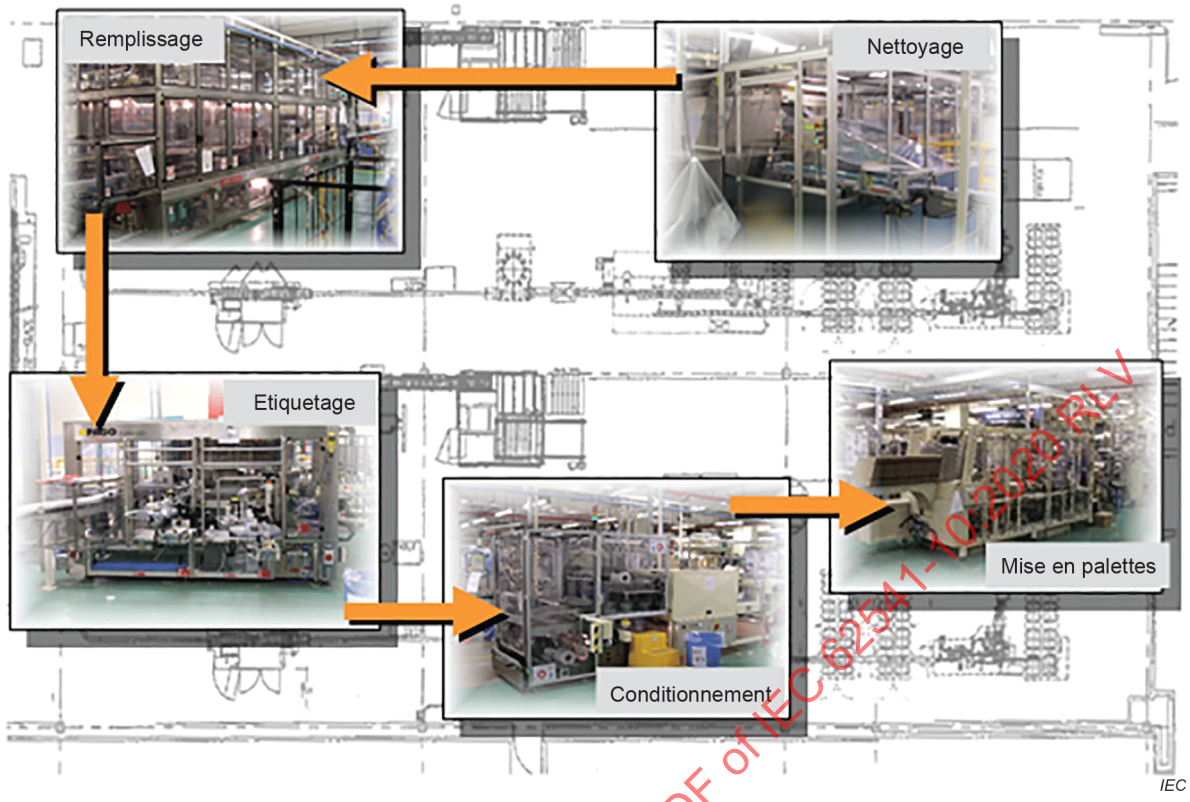


Figure 1 – Commande d'installation d'automatisation

Les *Fonctions* de modélisation des *Méthodes* et *Programmes* mises en œuvre dans les *Serveurs OPC* et les systèmes sous-jacents présentent généralement des domaines d'application, des comportements, des durées de vie et des niveaux de complexité différents. En règle générale, ces *Fonctions* ne sont pas caractérisées par les opérations de lecture ou d'écriture de données réalisées avec le jeu de services d'*Attributs* OPC UA.

Les *Méthodes* représentent les *Fonctions* de base du *Serveur* que le *Client* peut invoquer. En revanche, les *Programmes* permettent de modéliser une fonctionnalité dynamique plus complexe dans le système. Par exemple, un appel de méthode peut être utilisé pour réaliser un calcul ou réinitialiser un compteur. Un *Programme* est utilisé pour exécuter et commander un traitement par lots, exécuter un programme-pièce de machine-outil ou gérer un téléchargement de domaine. Les *Méthodes* et leur mécanisme d'invocation sont décrits dans l'IEC 62541-3 et l'IEC 62541-4.

Le présent document décrit les extensions aux capacités fondamentales définies dans l'IEC 62541-5, ou leur utilisation particulière, exigées pour les *Programmes*.

4.2 Programmes

4.2.1 Vue d'ensemble

Les *Programmes* sont des *Fonctions* complexes résidant dans un *Serveur* ou un système sous-jacent qu'un *Client* peut invoquer et gérer. Les *Programmes* peuvent représenter n'importe quel niveau de fonctionnalité au sein d'un système ou d'un processus pour lequel la commande ou l'intervention du *Client* est exigée et dont la surveillance du déroulement est souhaitée. Ce modèle est représenté à la Figure 2.

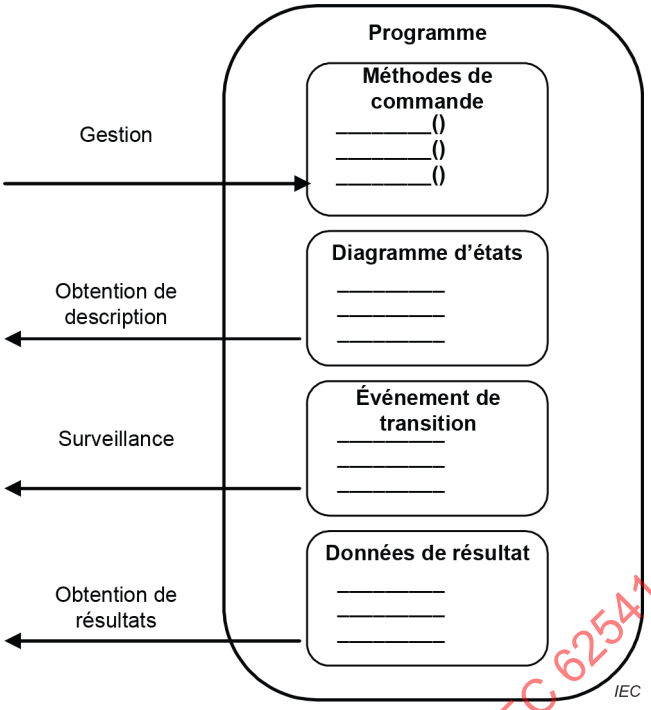


Figure 2 – Représentation d'un Programme

L'exécution des *Programmes* dynamiques se déroule par transition selon une séquence d'états prédéfinie. Le comportement des *Programmes* est défini par un *Diagramme d'états finis de programme (PFSM – Program Finite State Machine)*. Les éléments du PFSM décrivent les phases d'exécution d'un *Programme* en ce qui concerne les transitions valides entre un ensemble d'états, les stimuli ou causes de ces transitions et les effets qui résultent de celles-ci.

4.2.2 Considérations relatives à la sécurité

Dans la mesure où les *Programmes* peuvent être utilisés pour réaliser des algorithmes de commande avancés ou d'autres actions, il convient de limiter leur utilisation au personnel disposant des droits d'accès appropriés. Il est recommandé de générer les *AuditUpdateMethodEvents* pour permettre la surveillance du nombre de *Programmes* en cours d'exécution en plus de leur fréquence d'exécution.

4.2.3 Diagramme d'états finis de programme

Le Tableau 1 répertorie les états, transitions, causes et effets constitutifs du *Diagramme d'états finis de programme* qui est représenté à la Figure 3.

Tableau 1 – Diagramme d'états finis de programme

Nº	Dénomination de transition	Cause	De l'état	A l'état	Effet
1	HaltedToReady	Méthode Reset	Halted	Ready	Consigne l'Evénement/le Résultat de la Transition 1
2	ReadyToRunning	Méthode Start	Ready	Running	Consigne l'Evénement/le Résultat de la Transition 2
3	RunningToHalted	Méthode Halt ou Interne (Erreur)	Running	Halted	Consigne l'Evénement/le Résultat de la Transition 3
4	RunningToReady	Interne	Running	Ready	Consigne l'Evénement/le Résultat de la Transition 4
5	RunningToSuspended	Méthode Suspend	Running	Suspended	Consigne l'Evénement/le Résultat de la Transition 5
6	SuspendedToRunning	Méthode Resume	Suspended	Running	Consigne l'Evénement/le Résultat de la Transition 6
7	SuspendedToHalted	Méthode Halt	Suspended	Halted	Consigne l'Evénement/le Résultat de la Transition 7
8	SuspendedToReady	Interne	Suspended	Ready	Consigne l'Evénement/le Résultat de la Transition 8
9	ReadyToHalted	Méthode Halt	Ready	Halted	Consigne l'Evénement/le Résultat de la Transition 9

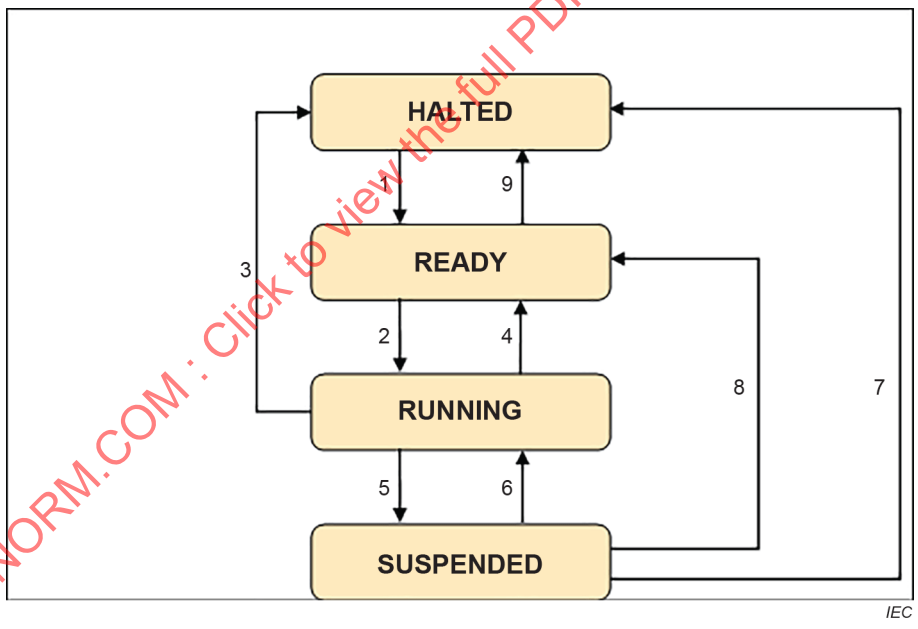


Figure 3 – Etats de Programme et transitions

4.2.4 Etats de Programme

Un ensemble normalisé d'états de base est défini pour les *Programmes*, faisant partie intégrante du *Diagramme d'états finis de programme.* Ces états représentent les stades auxquels un *Programme* peut exister à un moment donné, tel que visualisé par un *Client*. Cet état est l'état actuel du *Programme*. Tous les *Programmes* doivent prendre en charge cet ensemble de base. Un *Programme* peut ou non exiger une action du *Client* pour voir son état modifié. Le Tableau 2 définit les états de manière formelle.

Tableau 2 – Etats de Programme

Etat	Description
Ready	Le *Programme* est correctement initialisé et peut être démarré.
Running	Le *Programme* est en cours d'exécution jusqu'à son achèvement.
Suspended	Le *Programme* a été interrompu avant d'atteindre un état final, mais peut être repris.
Halted	Le *Programme* est parvenu à un état final ou d'échec; il ne peut pas être démarré ou repris sans réinitialisation.

L'ensemble des états définis pour décrire un *Programme* peut être étendu. Les sous-états de *Programme* peuvent être définis de sorte que les états de base fournissent une plus grande résolution d'un processus et décrivent la cause et les effets des stimuli et transitions supplémentaires. Les organismes de normalisation et les groupes industriels peuvent étendre le *Modèle d'états finis de programme* de base pour se conformer aux différents modèles industriels. Par exemple, l'état Halted peut comprendre les sous-états Aborted et Completed permettant d'indiquer si la *Fonction* a été achevée de manière satisfaisante avant de passer à la transition vers l'état Halted. Les états de transition tels que Starting ou Suspending peuvent également être des extensions de l'état Running, par exemple.

4.2.5 Transitions d'états

Un ensemble normalisé de transitions d'états est défini pour le *Diagramme d'états finis de programme*. Ces transitions définissent les modifications valides apportées à l'état actuel du *Programme*, par rapport à l'état initial et à l'état résultant. Le Tableau 3 définit les transitions de manière formelle.

Tableau 3 – Transitions d'états de Programme

N° de transition	Dénomination de transition	Etat initial	Etat résultant
1	HaltedToReady	Halted	Ready
2	ReadyToRunning	Ready	Running
3	RunningToHalted	Running	Halted
4	RunningToReady	Running	Ready
5	RunningToSuspended	Running	Suspended
6	SuspendedToRunning	Suspended	Running
7	SuspendedToHalted	Suspended	Halted
8	SuspendedToReady	Suspended	Ready
9	ReadyToHalted	Ready	Halted

4.2.6 Stimuli de transition d'états de Programme

Les stimuli ou les causes de transitions d'états d'un *Programme* peuvent être internes ou externes au *Serveur*. Les stimuli internes sont par exemple l'achèvement des étapes d'usinage, la détection d'une condition d'alarme ou la transmission d'un paquet de données. Les stimuli externes sont par exemple les *Méthodes*. Des *Méthodes* normalisées sont définies; elles agissent comme des stimuli pour commander un *Programme*.

4.2.7 Méthodes de Commande de Programme

Les *Clients* gèrent un *Programme* en appelant des *Méthodes*. Les *Méthodes* agissent sur le comportement d'un *Programme* en provoquant des transitions d'états spécifiées. Les transitions d'états régissent les actions réalisées par le *Programme*. Le présent document

définit un ensemble de *Méthodes de Commande de Programme* normalisées. Ces *Méthodes* fournissent les moyens suffisants au *Client* pour exécuter un *Programme*.

Le Tableau 4 répertorie l'ensemble des *Méthodes de Commande de Programme* définies. Chaque *Méthode* provoque des transitions à partir d'états spécifiés et doit être appelée lorsque le *Programme* est parvenu à l'un de ces états.

Des *Programmes* individuels peuvent en option prendre en charge n'importe quel sous-ensemble des *Méthodes de Commande de Programme*. Par exemple, certains *Programmes* peuvent ne pas être autorisés à déclencher un état d'interruption et ne fournissent de ce fait pas les *Méthodes Suspend* et *Resume*.

Les *Programmes* peuvent prendre en charge des *Méthodes* supplémentaires définies par l'utilisateur. Les *Méthodes* définies par l'utilisateur ne doivent pas modifier le comportement du *Diagramme d'états finis de programme* de base.

Tableau 4 – Méthodes de Commande de Programme

Dénomination de méthode	Description
Start	Fait passer le *Programme* de l'état Ready à l'état Running.
Suspend	Fait passer le *Programme* de l'état Running à l'état Suspended.
Resume	Fait passer le *Programme* de l'état Suspended à l'état Running.
Halt	Fait passer le *Programme* de l'état Ready, Running ou Suspended à l'état Halted.
Reset	Fait passer le *Programme* de l'état Halted à l'état Ready.

Toutes les *Méthodes de Commande de Programme* sont définies par leur *BrowseName* dans le *ProgramStateMachineType*, selon la *ModellingRule OptionalPlaceholder*. Comme précisé dans l'IEC 62541-3, cette règle permet l'ajout d'*Arguments* à ces *Méthodes* pour les sous-types et instances. Par exemple, une *Méthode* Start peut inclure un argument d'options qui spécifie des options dynamiques utilisées pour déterminer un comportement donné du programme. Le service d'*Appel* de *Méthode* spécifié dans l'IEC 62541-4 définit un statut de retour. Ce statut de retour indique la réussite de la *Méthode de Commande de Programme* ou la raison de son échec.

4.2.8 Effets des transitions d'états de Programme

La transition d'états d'un *Programme* a généralement une cause et produit également un effet. L'effet est un sous-produit de la transition d'états d'un *Programme*, qu'un *Client* peut utiliser pour surveiller l'avancement du *Programme*. Les effets peuvent être internes ou externes. Un effet externe d'une transition d'états est caractérisé par la génération d'une notification d'*Evénement*. Chaque transition d'états de *Programme* est associée à un *Evénement* unique. Ces *Evénements* reflètent la progression et la trajectoire du *Programme* dans son ensemble d'états définis. Les effets internes d'une transition d'états peuvent se caractériser par la réalisation d'une action de programmation donnée telle que la génération de données.

4.2.9 Données de résultat de Programme

4.2.9.1 Vue d'ensemble

Les données de résultat sont générées par un *Programme* en cours d'exécution. Les données de résultat peuvent être intermédiaires ou finales. Les données de résultat peuvent être associées à des transitions d'états de *Programme* spécifiques.

4.2.9.2 Données de résultat intermédiaires

Les données de résultat intermédiaires sont transitoires et sont générées par le *Programme* conjointement avec les transitions d'états non finales. Les éléments de données constitutifs des résultats intermédiaires sont définis en association avec les transitions d'états de *Programme* spécifiques. Leurs valeurs ne sont significatives qu'au niveau de la transition.

Chaque transition d'états de *Programme* peut être associée à différents éléments de données de résultat. A l'inverse, un ensemble de transitions peut partager un élément de données de résultat. Le pourcentage d'achèvement est un exemple de données de résultat intermédiaires. La valeur du pourcentage d'achèvement est obtenue lorsque la transition d'états se produit et est disponible pour le *Client*.

Les *Clients* acquièrent les données de résultat intermédiaires en s'abonnant aux *Evénements* de transition d'états de *Programme*. Les *Evénements* spécifient les éléments de données pour chaque transition. Lorsque la transition se produit, l'*Evénement* généré transmet les valeurs de données de résultat obtenues aux *Clients* abonnés. Si aucun *Client* ne surveille le *Programme*, les données de résultat intermédiaires peuvent être éliminées.

4.2.9.3 Données de résultat finales

Les données de résultat finales sont générées à la fin de l'exécution du *Programme*. Le temps d'exécution total, le nombre de widgets produits et la condition de défaut rencontrée sont des exemples de données de résultat finales. Lorsque le *Programme* passe à l'état final, ces données de résultat peuvent être transmises au *Client* par l'*Evénement* de transition. Les données de résultat finales sont également disponibles dans le *Programme* pour être lues par un *Client* après l'arrêt du programme. Ces données sont conservées jusqu'à une nouvelle exécution ou jusqu'à la suppression de l'Instance du *Programme*.

4.2.9.4 Surveillance des Programmes

Les *Clients* peuvent surveiller les activités associées à l'exécution d'un *Programme*. Ces activités comprennent l'invocation des *Méthodes* de gestion, la génération de données de résultat et la progression du *Programme* dans ses états. Les *Evénements d'Audit* sont fournis par les *Appels de Méthode* et les transitions d'états. Ces *Evénements* permettent de conserver un enregistrement des *Clients* qui interagissent avec un *Programme* et des transitions d'états du *Programme* résultant de cette interaction.

4.2.10 Durée de vie d'un Programme

4.2.10.1 Vue d'ensemble

Les *Programmes* peuvent avoir différentes durées de vie. Certains *Programmes* peuvent résider de manière permanente sur un *Serveur*, tandis que d'autres sont créés et retirés. La création et le retrait peuvent être commandés par un *Client* ou peuvent être limités à des moyens locaux.

Un *Programme* peut pouvoir être créé par le *Client*. Dans ce cas, le *Client* peut ajouter le *Programme* au *Serveur*. La *Méthode de Création d'Objet* définie dans l'IEC 62541-3 permet de créer l'instance du *Programme*. L'état initial du *Programme* peut être Halted ou Ready. Certains *Programmes*, par exemple, peuvent exiger la mise à disposition d'une ressource après leur création, avant d'être prêts pour l'exécution. Dans ce cas, le *Programme* est initialisé à l'état Halted pour ensuite passer à l'état Ready lorsque la ressource est fournie.

Un *Programme* peut pouvoir être retiré par le *Client*. Dans ce cas, le *Client* peut supprimer l'instance du *Programme* du *Serveur*. Le *Service DeleteNodes* défini dans l'IEC 62541-4 permet de retirer l'instance du *Programme*. Le *Programme* doit être à l'état Halted pour être retiré. Un *Programme* peut également pouvoir être retiré automatiquement. Dans ce cas, le *Programme* se supprime lui-même à la fin de son exécution.

4.2.10.2 Instances d'un Programme

Les *Programmes* peuvent avoir une ou plusieurs instances. Un *Serveur* peut prendre en charge plusieurs instances d'un *Programme* si elles peuvent être exécutées en parallèle. Par exemple, le *Programme* peut définir une *Méthode Start* présentant un argument d'entrée destiné à spécifier la ressource qui est utilisée par ses *Fonctions*. Chaque instance du *Programme* est ensuite démarrée en désignant l'utilisation des différentes ressources. Le *Client* peut découvrir toutes les instances d'un *Programme* en cours d'exécution sur un *Serveur*. Chaque instance d'un *Programme* est identifiée de manière unique sur le *Serveur* et est gérée de manière indépendante par le *Client*.

4.2.10.3 Recyclage d'un Programme

Les *Programmes* peuvent être exécutés une ou plusieurs fois (recyclés). Un *Programme* exécuté une fois reste indéfiniment à l'état Halted après son exécution. La procédure normale consiste à le supprimer après examen de ses résultats finaux.

Les *Programmes* recyclables peuvent comporter un nombre de cycles limité ou illimité. Ces *Programmes* peuvent exiger une étape de réinitialisation pour passer de l'état Halted à l'état Ready. Cela permet de reconstituer les ressources ou de réinitialiser les paramètres avant de redémarrer le *Programme*. La *Méthode de Commande de Programme Reset* déclenche cette transition d'états et toute action ou tout effet associé.

5 Modèle

5.1 Généralités

Le modèle de *Programme* étend le modèle *FiniteStateMachineType* et le modèle de base *ObjectType* présentés dans l'IEC 62541-5. Chaque *Programme* comporte une *Définition de Type* qui constitue le sous-type du *FiniteStateMachineType*. Le *ProgramStateMachineType* décrit le modèle de *Diagramme d'états finis* pris en charge par toute *Invocation de Programme* du type en question. Le *ProgramStateMachineType* définit également le jeu de propriétés qui caractérise les aspects particuliers du comportement du *Programme*, tels que la durée de vie et le recyclage, ainsi que les données de résultat générées par le *Programme*.

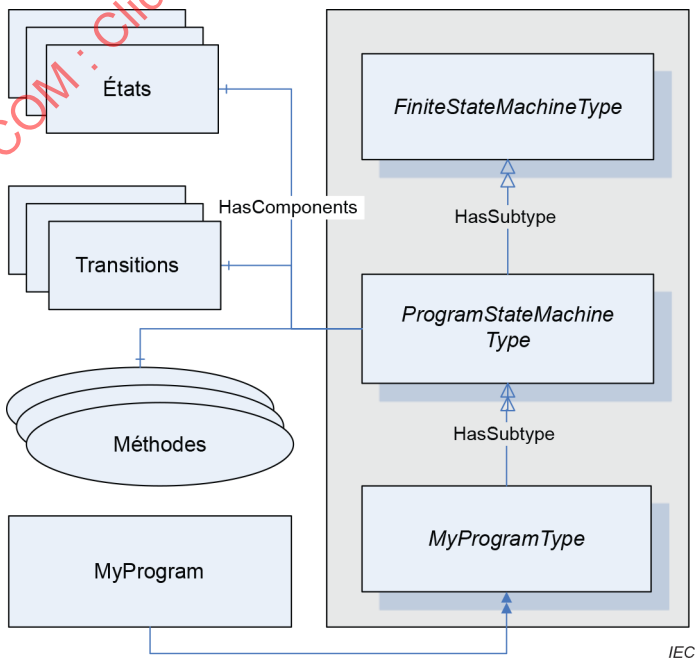


Figure 4 – Type de programme

Le *ProgramStateMachineType* de base définit le *Diagramme d'états finis* normalisé spécifié pour tous les *Programmes*. Cela comprend les états, les transitions et les causes (*Méthodes*) et les effets (*Evénements*) des transitions. Les sous-types du *ProgramStateMachineType* de base peuvent être définis pour étendre ou caractériser de manière plus spécifique le comportement d'un *Programme* individuel, comme représenté par "MyProgramType" à la Figure 4.

5.2 ProgramStateMachineType

5.2.1 Vue d'ensemble

Le Tableau 5 répertorie les propriétés et composants supplémentaires qui constituent le *ProgramStateMachineType*. Aucune sémantique spécifique au *ProgramStateMachineType* n'est attribuée aux autres *ObjectTypes* de base, *Attributs FiniteStateMachineType* ou *Propriétés*.

Tableau 5 – ProgramStateMachineType

Attribut	Valeur				
	Comprend tous les attributs spécifiés pour le FiniteStateMachineType				
BrowseName	ProgramStateMachineType				
IsAbstract	False				
Références	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	Creatable	Boolean	PropertyType	–
HasProperty	Variable	Deletabe	Boolean	PropertyType	Obligatoire
HasProperty	Variable	AutoDelete	Boolean	PropertyType	Obligatoire
HasProperty	Variable	RecycleCount	Int32	PropertyType	Obligatoire
HasProperty	Variable	InstanceCount	UInt32	PropertyType	–
HasProperty	Variable	MaxInstanceCount	UInt32	PropertyType	–
HasProperty	Variable	MaxRecycleCount	UInt32	PropertyType	–
HasComponent	Variable	ProgramDiagnostic	Program Diagnostic2 DataType	Program Diagnostic2Type	Facultative
HasComponent	Objet	Halted		StateType	–
HasComponent	Objet	Ready		StateType	–
HasComponent	Objet	Running		StateType	–
HasComponent	Objet	Suspended		StateType	–
HasComponent	Objet	HaltedToReady		TransitionType	–
HasComponent	Objet	ReadyToRunning		TransitionType	–
HasComponent	Objet	RunningToHalted		TransitionType	–
HasComponent	Objet	RunningToReady		TransitionType	–
HasComponent	Objet	RunningToSuspended		TransitionType	–
HasComponent	Objet	SuspendedToRunning		TransitionType	–
HasComponent	Objet	SuspendedToHalted		TransitionType	–
HasComponent	Objet	SuspendedToReady		TransitionType	–
HasComponent	Objet	ReadyToHalted		TransitionType	–
HasComponent	Méthode	Start			Optional Placeholder
HasComponent	Méthode	Suspend			Optional Placeholder
HasComponent	Méthode	Reset			Optional Placeholder
HasComponent	Méthode	Halt			Optional Placeholder
HasComponent	Méthode	Resume			Optional Placeholder
HasComponent	Objet	FinalResultData		BaseObjectType	Facultative

5.2.2 Propriétés ProgramStateMachineType

La *Propriété Creatable* est un booléen qui indique si les *Invocations de Programme* de ce *ProgramStateMachineType* peuvent être créées par un *Client*. Si la valeur est False, ces *Invocations de Programme* sont persistantes ou peuvent uniquement être créées par le *Serveur*.

La *Propriété Deletable* est un booléen qui indique si une *Invocation de Programme* de ce *ProgramStateMachineType* peut être supprimée par un *Client*. Si la valeur est False, ces *Invocations de Programme* peuvent uniquement être supprimées par le *Serveur*.

La *Propriété AutoDelete* est un booléen qui indique si les *Invocations de Programme* de ce *ProgramStateMachineType* sont retirées par le *Serveur* à la fin de l'exécution. Si la valeur est False, ces *Invocations de Programme* demeurent sur le *Serveur* jusqu'à leur suppression par le *Client*. Lorsque l'*Invocation de Programme* est supprimée, toutes les données de résultat associées à l'instance sont également retirées.

La *Propriété RecycleCount* est un entier non signé qui indique le nombre de fois qu'une *Invocation de Programme* de ce type a été recyclée ou redémarrée depuis son démarrage (et non reprise). Noter que la Méthode Reset peut être exigée pour préparer le redémarrage d'un *Programme*.

La *Propriété MaxRecycleCount* est un entier qui indique le nombre maximal de fois qu'une *Invocation de Programme* de ce type peut être recyclée ou redémarrée depuis son démarrage (et non reprise). Si la valeur est inférieure à 0, le nombre de redémarrages est illimité. Si la valeur est égale à zéro, le Programme ne peut être recyclé ni redémarré.

La *Propriété InstanceCount* est un entier non signé qui indique le nombre d'*Invocations de Programme* de ce type qui existent actuellement.

La *Propriété MaxInstanceCount* est un entier qui indique le nombre maximal d'*Invocations de Pogramme* de ce type qui peuvent exister simultanément sur ce *Serveur*. Si la valeur est inférieure à 0, ce nombre est illimité.

5.2.3 Composants de ProgramStateMachineType

5.2.3.1 Vue d'ensemble

Les composants de *ProgramStateMachineType* comprennent un jeu de *Références* aux instances d'*Objet* de *StateTypes*, *TransitionTypes*, *EventTypes* et les *Méthodes* qui, dans leur ensemble, définissent le *FiniteStateMachine* du *Programme*.

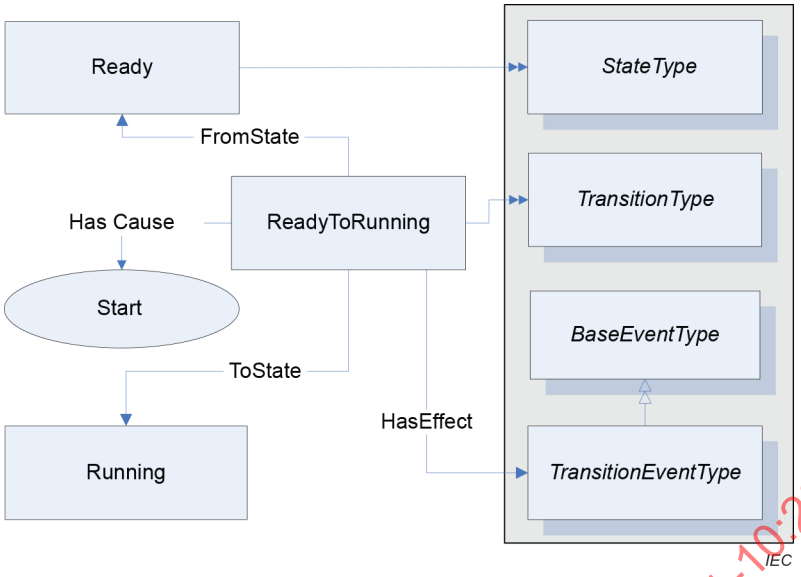


Figure 5 – Références FSM de Programme

La Figure 5 représente les *Références* de composants qui définissent les associations entre deux des états de *ProgramStateMachineType*, Ready et Running. Les ReferenceTypes supplémentaires ont été omis pour simplifier la représentation.

5.2.3.2 Etats de ProgramStateMachineType

Le Tableau 6 spécifie les *Objets* d'états de *ProgramStateMachineType*. Ces *Objets* sont des instances du *StateType* défini dans l'IEC 62541-5. Une valeur *StateNumber* unique est attribuée à chaque état. Les sous-types du *ProgramStateMachineType* peuvent ajouter des références de n'importe quel état à un *Objet StateMachine* subordonné ou imbriqué pour étendre le *FiniteStateMachine*.

Tableau 6 – Etats de Programme

BrowseName	Références	BrowseName Cible	Valeur	TypeDefinition Cible	NOTES
Etats					
Halted	HasProperty	StateNumber	11	PropertyType	
	ToTransition	HaltedToReady		TransitionType	
	FromTransition	RunningToHalted		TransitionType	
	FromTransition	SuspendedToHalted		TransitionType	
	FromTransition	ReadyToHalted		TransitionType	
Ready	HasProperty	StateNumber	12	PropertyType	
	FromTransition	HaltedToReady		TransitionType	
	ToTransition	ReadyToRunning		TransitionType	
	FromTransition	RunningToReady		TransitionType	
	ToTransition	ReadyToHalted		TransitionType	
Running	HasProperty	StateNumber	13	PropertyType	
	ToTransition	RunningToHalted		TransitionType	
	ToTransition	RunningToReady		TransitionType	
	ToTransition	RunningToSuspended		TransitionType	
	FromTransition	ReadyToRunning		TransitionType	
	FromTransition	SuspendedToRunning		TransitionType	
Suspended	HasProperty	StateNumber	14	PropertyType	
	ToTransition	SuspendedToRunning		TransitionType	
	ToTransition	SuspendedToHalted		TransitionType	
	ToTransition	SuspendedToReady		TransitionType	
	FromTransition	RunningToSuspended		TransitionType	

L'état Halted est l'état inactif d'un *Programme*. Il peut s'agir d'un état initial ou d'un état final. S'il s'agit d'un état initial, l'*Invocation de Programme* ne peut pas commencer l'exécution du fait des conditions existant sur le *Serveur*. S'il s'agit d'un état final, l'état Halted peut indiquer un état d'échec ou d'achèvement du *Programme*. Un état ou résultat subordonné peut être utilisé pour déterminer la nature de l'arrêt du programme. L'état Halted fait référence à quatre *Objets de Transition*, qui identifient les transitions d'états admises vers l'état Ready et depuis les états Ready, Running et Suspended.

L'état Ready indique que le *Programme* est prêt à commencer l'exécution. Les *Programmes* qui sont prêts à commencer l'exécution dès leur création peuvent immédiatement passer à l'état Ready. L'état Ready fait référence à quatre *Objets de Transition*, qui identifient les transitions d'états admises vers les états Running et Halted et depuis les états Halted et Ready.

L'état Running indique que le *Programme* est en cours de réalisation de sa *Fonction*. L'état Running fait référence à cinq *Objets de Transition*, qui identifient les transitions d'états admises vers les états Halted, Ready et Suspended et depuis les états Ready et Suspended.

L'état Suspended indique que le *Programme* a arrêté la réalisation de sa *Fonction*, mais conserve la capacité de reprendre la *Fonction* au point auquel elle a été interrompue. L'état

Suspended fait référence à quatre *Objets de Transition*, qui identifient les transitions d'états admises vers les états Ready, Running et Halted et depuis l'état Ready.

5.2.3.3 Transitions de ProgramStateMachineType

Les transitions de *ProgramStateMachineType* sont des instances du *TransitionType* défini dans l'IEC 62541-5, qui comprend également les définitions des références ToState, FromState, HasCause et HasEffect utilisées. Le Tableau 7 spécifie les transitions définies pour le *ProgramStateMachineType*. Un *TransitionNumber* unique est attribué à chaque transition. La colonne "Notes" indique lorsqu'une cause fait référence à des *Méthodes* et lorsque les effets sont facultatifs.

Tableau 7 – Transitions de Programme

BrowseName	Références	BrowseName Cible	Valeur	TypeDefinition Cible	Notes
Transitions					
HaltedToReady	HasProperty	TransitionNumber	1	PropertyType	
	ToState	Ready		StateType	
	FromState	Halted		StateType	
	HasCause	Reset			Méthode
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
ReadyTo Running	HasProperty	TransitionNumber	2	PropertyType	
	ToState	Running		StateType	
	FromState	Ready		StateType	
	HasCause	Start			Méthode
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
RunningTo Halted	HasProperty	TransitionNumber	3	PropertyType	
	ToState	Halted		StateType	
	FromState	Running		StateType	
	HasCause	Halt			Méthode
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
RunningTo Ready	HasProperty	TransitionNumber	4	PropertyType	
	ToState	Ready		StateType	
	FromState	Running		StateType	
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			

BrowseName	Références	BrowseName Cible	Valeur	TypeDefinition Cible	Notes
RunningTo Suspended	HasProperty	TransitionNumber	5	PropertyType	
	ToState	Running		StateType	
	FromState	Suspended		StateType	
	HasCause	Suspend			Méthode
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
SuspendedTo Running	HasProperty	TransitionNumber	6	PropertyType	
	ToState	Running		StateType	
	FromState	Suspended		StateType	
	HasCause	Resume			Méthode
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
SuspendedTo Halted	HasProperty	TransitionNumber	7	PropertyType	
	ToState	Halted		StateType	
	FromState	Suspended		StateType	
	HasCause	Halt			Méthode
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
SuspendedTo Ready	HasProperty	TransitionNumber	8	PropertyType	
	ToState	Ready		StateType	
	FromState	Suspended		StateType	
	HasCause	Reset			Méthode
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			
ReadyToHalted	HasProperty	TransitionNumber	9	PropertyType	
	ToState	Halted		StateType	
	FromState	Ready		StateType	
	HasCause	Halt			Méthode
	HasEffect	ProgramTransitionEventType			
	HasEffect	AuditProgramTransitionEventType			

La transition *HaltedToReady* spécifie la transition de l'état Halted à l'état Ready. Elle peut être déclenchée par la *Méthode Reset*.

La transition *ReadyToRunning* spécifie la transition de l'état Ready à l'état Running. Elle est déclenchée par la *Méthode Start*.